



FutureBASIC

Reference

© 2004 STAZ Software

STAZ Software, Inc.
7582 Alakoko Drive
Diamondhead, MS 39525

Technical support:

Voice: (228) 255-7085 (Monday through Friday, 9 AM to 5 PM Central)

Fax: (228) 255-7086

<mailto:tech@stazsoftware.com>

<http://www.stazsoftware.com>

FutureBASIC3 and FB^3 are trademarks of STAZ Software, Inc. All other products and logos mentioned in this documentation are trademarks of their respective owners.

Special thanks to Alain Pastor for his continued policing and extensive help on this ever-expanding manuscript. Without his help, we would never have been able to conquer this beast.



Introduction

Introduction to the FB^3 Language

An FB^3 program consists of a series of statements, and optional statement labels. Usually you will write each statement on a separate line; however, it's possible to put multiple statements on a single line, and conversely a single statement may span multiple lines:

- You can write multiple statements on a single line by separating each pair of statements with a colon, as in this example:

```
A = 7 * B : Print score(A) : Fn doSum(x)
```

Note that when the preference for QuickBASIC labels is turned on (See the Editor manual) a statement may look like a QuickBASIC label to the compiler. Example:

```
Print:Stop:Rem this won't work with QB labels On
```

Because QuickBASIC labels require that no space be present between the text and the colon, you can side step possible errors as follows:

```
Print : Stop : Rem note the space beFore the colon
```

- If a line ends with the *continuation character*, the (last) statement on that line continues on the next line. The continuation character looks like this: `↵`, and it is typed by entering Option-L. Example:

```
myArray(numberDogs, numberCats, numberPorcupines) ↵
    = Fn createMenagerie(Mid$(animalName$(x), ↵
                        nameOffset, 4), vetNumber)
```

While you can safely insert a line-break character [option-L] in a quoted string, elsewhere it is wise to only insert line-break characters between statements, and definitely not to mix them with block statement. For those who still use line numbers, Any line in your program (except lines which follow a continuation character) may begin with a line number.

Certain special statements mark the beginning or end of a “block structure.” A statement which opens or closes a block structure should not appear with any other statement on the same line. Such statements include the following:

#If	#EndIf	#Else
Begin Enum	End Enum	
Begin Globals	End Globals	
Begin Record	End Record	
BeginAssem	EndAssem	
Compile Long If	Compile End If	Compile Xelse
Dim Record	Dim End Record	
Do	Until	
EnterProc	ExitProc	
EnterProc%	ExitProc%	
For	Next	
Local Fn	End Fn	
Long Fn	End Fn	
Long If	End If	Xelse
Select Case	End Select	Case
While	Wend	

Line Numbers

Any line in your program (except lines which follow a continuation character) may begin with a line number. A line number must be an integer in the range 1 through 65534, and no two lines may have the same number. At least one space character should separate the line number from the first statement in the line. Example:

```
140 Print "Hello": Beep
```

Line numbers can be useful for identifying a particular location in your program, as a target for statements such as `Goto` or `Gosub`. However, line numbers are considered an antiquated syntax feature, and their use is generally discouraged.

Statement Labels

A statement label is another way to identify a particular location in your program. A statement label must appear on a line by itself, and the same label can't appear more than once in your program. Depending on your Preferences settings, you can express a statement label in the following ways:

- A string of characters inside double-quotes. Example:

```
"My First Label"
```

- A string of non-space characters followed by a colon. This kind of label is only available if the "Allow QuickBasic Labels" preference is set. Example:

```
Label17:
```

Like line numbers, statement labels can be useful if your program contains statements like `Goto` or `Gosub`; or if your program uses the `Line` or `Proc` functions.

Executable and Non-executable Statements

Every statement in FB³ is either an executable statement or a non-executable statement.

- Executable statements represent instructions which are to be performed when the program is run. When you run your program, a given executable statement may be performed once, or more than once, or not at all, depending on the program conditions. The order in which executable statements are performed is not necessarily the same as the order in which they appear in your program. Examples of executable statements are: `Print`; `Let`; `Read`; `Open`.
- Non-executable statements represent instructions which tell the FB³ compiler how to build your program. They help the compiler to determine how to allocate memory, and how to interpret and compile other statements in your program. The order in which non-executable statements appear in your program is important: when the compiler builds your program, it scans all statements from top to bottom, and a non-executable statement can only affect the interpretation of lines which appear below it. You cannot change the effect of a non-executable statement by putting it inside a "conditional execution" block such as `Long If...End If`, or `For...Next`. However, you can conditionally include or exclude a non-executable statement from the program by putting it inside a `Compile Long If` block. Examples of non-executable statements are: `Dim`; `Data`; `Local Fn`; `Def Len`.

Program Layout and Order of Execution

Parts of your program may be located within function blocks and procedure blocks. Function and procedure blocks are blocks of statements which are surrounded by the following pairs of statements:

```
Local Fn...End Fn
Long Fn...End Fn
EnterProc...ExitProc
```

Statements which are outside of all function and procedure blocks are said to belong to the “main” part of your program. “Main” statements may either precede function & procedure blocks, or follow them, or both.

When your program runs, the first executable statement that appears in “main” is the first statement that actually executes. After that, the order of execution depends on the specifics of the program: normally, statements are executed in the order in which they appear in “main,” but the flow of execution can be altered by statements like `Long If; Goto; While;` `Fn <userFunction>;` etc. Function blocks and procedure blocks are not executed until they are explicitly “called,” using a statement like `Fn <userFunction>`. The program stops when it reaches an `End` or `Stop` statement, or when it reaches the last statement in “main.” Note that if there are no executable statements in “main,” your program won’t do anything.

Identifiers

An identifier is a name that you make up to identify something in your program. In FB^3, identifiers are used to name the following kinds of things:

- variables (see Appendix B);
- arrays (see below);
- user-defined functions (see the `Local Fn`, `Long Fn` etc. statements);
- user-defined data types (see the `#Define` and `Begin Record` statements);
- record fields (see the `Begin Record` and `Dim` statements);
- symbolic constants (see the “Constant declaration” statement).

In FB^3, an identifier can have any length up to 245 characters. It must start with a letter; its subsequent characters can be any combination of letters, numeric digits, and underscore characters (`_`) (symbolic constant names should not contain embedded underscores). Identifiers which represent variables, arrays and function names may also be followed by a “type-identifier suffix,” which is a 1- or 2-character symbol that specifies the item’s data type (see Appendix C for a list of type-identifier suffixes). Identifiers are “case-insensitive,” so the identifiers “highscore” and “HighScore” are both recognized as the same identifier.

You can use the same identifier to name two different things, as long as the context makes it clear which thing is being referred to. For example, your program may have a variable identified as “BookList\$”, and also a local function identified as “BookList\$”; FB^3 can distinguish between these two, because references to local functions are always preceded by the `Fn` keyword in your program.

Variables

A variable can be thought of as a “named container for data.” There are a number of different ways to represent variables in FB^3; see Appendix B for more information. Every variable is associated with a “data type,” which determines the amount of memory that the variable occupies, and how the variable’s value is interpreted. There are a number of data types pre-defined in FB^3; you can also create user-defined data types called records. See Appendix C for more information.

Arrays

An array is a collection of variables which all share the same name and same data type. Each variable in the collection is called an element of the array; your program distinguishes one element from another by means of subscripts, which are numbers in parentheses following the array’s name. For example, if “theAngle” is the name of an array, then “theAngle(3)” represents one element of the array, and “theAngle(4)” represents a different element of the array.

The example above illustrates a one-dimensional array. In a two-dimensional array, each element is represented by a unique ordered pair of subscripts. If “Salary&” is the name of a two-dimensional array, then “Salary&(4,7)” represents an element of the array, and “Salary&(7,4)” represents a different element of the array.

An array can have more than two dimensions; in fact, it can have up to 255. You use the `Dim` statement or the `Xref[@]` to declare how many dimensions a given array has, and to specify the maximum values that can be assigned to each subscript. The minimum value that can be assigned to a subscript is always 0.

Code Size

In days past, the 68K processor imposed a 32K limit per code segment on code size. This forced programmers to slice and dice applications to hold each segments inside of this margin. With the adoption of the PPC processor, code size is now limited to the maximum allowable size of a JMP instruction, 24 megabytes.

Conventions Used in this Manual

In the syntax descriptions that appear in the remainder of this manual, the following conventions apply:

- Items in *italics* represent placeholders which should be replaced as indicated in the description;
- Items in **bold text** represent literal text that you should enter exactly as shown;
- Items in plain non-italic text represent literal text that you should usually enter exactly as shown; however, the following characters should not be entered, but have special meanings explained below:
`[] { } | ...`
- Items enclosed in square brackets `[]` are optional;
- Items separated by vertical bars `|` and enclosed by curly brackets `{ }` represent a list from which one item should be chosen;
- Items separated by vertical bars `|` and enclosed by square brackets `[]` represent a list from which one or zero items should be chosen;
- An ellipsis (...) indicates that the preceding item may be repeated an indefinite number of times.

Example: Consider the following syntax description template:

```
bob [, {bill | ron [, rick]}]
```

This template matches each of the following:

```
bob
bob, bill
bob, ron
bob, ron, rick
```




Reference

#Define statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
#Define newTypeName As oldTypeName
#Define newTypeName As {Pointer To|@|^|.} oldTypeName
#Define newTypeName As {Handle To|@@|^|^|..} oldTypeName
```

Description:

The `#Define` statement is one way to create a name for a variable type (the other way to do so is to use the `Begin Record` statement). *newTypeName* can be any new name you like that is different from the names of all existing types. *oldTypeName* is the name of an existing type; this can either be a built-in type such as `Rect` or `Int`, or a type which you created previously, in a `Begin Record` statement or in another `#Define` statement. After the `#Define` statement, you can declare variables of the new type using `Dim` statements, and you can pass *newTypeName* to the `SizeOf` and `TypeOf` functions.

If you use the first syntax, *newTypeName* essentially becomes a synonym for *oldTypeName*. If you use the other two syntaxes, then variables of the new type are recognized as pointers or handles to structures of *oldTypeName*. This is the only way to create a type name for pointers or handles to other types.

Note:

`#Define` is non-executable, so you can't change its effect by putting it inside a conditional execution structure such as `Long If...End If`.

A non-executable statement inside a `Compile Long If` block will only be compiled if the condition following the `If` is met. Otherwise it will be ignored.

See Also:

`Begin Record`, `SizeOf`, `TypeOf`, `Dim`

#Else	statement	
✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>

Syntax:

```
#Else
```

Description:

#Else is a synonym for `Compile Xelse`. If you use the #Else statement, you must also use the #If and #EndIf statements.

See Also:

`Compile Long If`, `Compile Xelse`, `#If`, `#EndIf`

#EndIf statement

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

`#EndIf`

Description:

`#EndIf` is a synonym for `Compile End If`. If you use the `#EndIf` statement, you must also use the `#If` statement.

See Also:

`Compile Long If`; `Compile End If`; `#If`; `#Else`

#If **statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

#If condition
    [statementBlock1]
[#Else
    [statementBlock2]]
#EndIf

```

Description:

#If is a synonym for Compile Long If. If you use the #If statement, you must also use the #EndIf statement.

See Also:

Compile Long If; #Else; #EndIf

@Fn**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
functionAddress& = @Fn functionName
```

Description:

Returns a memory address which can be used to access the function specified by *functionName*. The *functionName* must be the name of a function which was defined or prototyped at some earlier location in the source code (in a Local Fn, a Long Fn, a Def Fn <expr>, or a Def Fn <proTOTYPE> statement).

The address returned by @Fn can be used in the Def Fn Using statement, or in the Fn Using statement/function.

See Also:

```
Def Fn Using; Line
```


Abs function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
positiveValue = Abs(expr)
```

Description:

The `Abs` function returns the absolute value of the numeric expression *expr*.

The absolute value of a number is its distance from zero. Thus, the number 3 has an absolute value of 3, while the number -12.34 has an absolute value of 12.34. The absolute value of zero is zero.

Example:



CD Example: Abs.bas

Note:

The standard `Abs` function can accept either an integer or a floating-point *expr* parameter.

See Also:

Usr Abs

Acos**function**

✓ *Appearance***✓** *Standard***✓** *Console*

Syntax:

```
radianAngle# = Acos(expr)
```

Description:

Returns the arccosine of *expr* in radians. In other words, if *expr* represents the cosine of some angle, then `Acos(expr)` returns the angle. The returned angle will be in the range of 0 to π radians (which corresponds to 0 to 180 degrees). `Acos` always returns a double-precision result.

See Also:

Sin; Cos; Tan; Atn; Asin

Acosh

function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

`result# = Acosh(expr)`

Description:

Returns the inverse hyperbolic cosine of `expr`. This is the (positive-valued) inverse of the `Cosh` function, so that `Acosh (Cosh (x))` equals `Abs (x)`. `Acosh` always returns a double-precision result.

See Also:

`Cosh`; `Asinh`; `Atanh`

And operator

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
result& = exprA {And | &&} exprB
```

Description:

Expression *exprA* and expression *exprB* are each interpreted as 32-bit integer quantities. The **And** operator performs a “bitwise comparison” of each bit in *exprA* with the bit in the corresponding position in *exprB*. The result is another 32-bit quantity; each bit in the result is determined as follows:

<i>Bit Value in exprA</i>	<i>Bit Value in exprB</i>	<i>Bit Value in resultat&</i>
0	0	0
1	0	0
0	1	0
1	1	1

The **And** operator can also be used to join two “condition clauses” for use in statements like **If**, **While** and **Until**. For example:

```
If n > 17 And myName$ <> "Smith" Then Beep
```

This statement produces a beep if and only if both *n > 17* is true and *myName\$ <> "Smith"* is true.

Even when it’s used to join condition clauses, the **And** operator still does a “bitwise comparison.” This happens because FB^3 actually assigns a numeric value to every condition clause, depending on whether the clause is true or false. For example, the clause *n > 17* is evaluated as -1 if it’s true, or as 0 if it’s false. Conversely, a numeric expression is judged as “true” if it’s non-zero, or as “false” if it’s zero.

Example:

In the following example, expressions are evaluated as true or false before a decision is made for branching. The logical expression *time > 7* is true, and is therefore evaluated as -1 . The expression *time < 8.5* is false, and is therefore evaluated as 0 . Then the bitwise comparison $(-1) \text{ And } (0)$ is performed, resulting in zero. Finally, the **Long If** statement interprets this zero result as meaning “false,” and therefore skips the first **Print** statement.

```
time = 9.5
Long If time > 7 And time < 8.5
    Print "It is time For breakfast!"
Xelse
    Print "We have To wait 'til noon To eat!"
End If
```

The example below shows how bits are manipulated with `And`:

```
DefStr Long
Print Bin$(923)
Print Bin$(123)
Print "-----"
Print Bin$(923 And 123)
```

Program output:

```
00000000000000000000000000000000000000001110011011  
0000000000000000000000000000000000000000000001111011  
-----  
00000000000000000000000000000000000000000000011011
```

Note:

In a statement like `If expr1 And expr2 Then...`, it is possible for “*expr1* And *expr2*” to be false even though each individual *expr* is evaluated as true. Consider this example:

```
JoeIsHere = 16
FredIsHere = 2
If JoeIsHere Then Print "Joe's here" Else Print "Joe's gone"
If FredIsHere Then Print "Fred's here"
    Else Print "Fred's gone"
Long If JoeIsHere And FredIsHere
    Print "They're both here"
Xelse
    Print "They're not both here!"
End If
```

Program output:

```
Joe's here
Fred's here
They're Not both here!
```

This strange result happens because the expression “16 And 2” evaluates to 0, which is then interpreted as “false” by the `Long If` statement. This wouldn’t have happened if we had set `JoeIsHere` to -1 and `FredIsHere` to -1, because the expression “1 And -1” evaluates to -1.

See Also:

Nand, Nor, Not; Xor; Or; Appendix D: *Numeric Expressions*

Annuity function

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
annuityFactor# = Fn Annuity#(rate, periods)
```

Description:

Returns the annuity factor for the given interest rate and number of periods. The interest rate should be expressed as a fraction of 1; for example, 5.2 percent should be expressed as 0.052.

Note:

Annuity uses the following formula:

$$\text{annuityFactor\#} = \frac{1 - (1 + \text{rate})^{-\text{periods}}}{\text{rate}}$$

See Also:

Compound

Appearance Button

statement

✓ *Appearance*✗ *Standard*✗ *Console*

Syntax:

```
Appearance Button [#] [-] id&[, [state][, [value][, -
[ min][, [max][, [title$][, [rect][, [type]]]]]]]
```

Revision:

February, 2002 (Release 6)

Description:

The `Appearance Button` statement puts a new control in the current output window, or alters an existing control's characteristics. After you create a button using the `Appearance Button` statement, you can use the `Dialog` function to determine whether the user has clicked it. You can use the `Button Close` statement if you want to dispose of the button without closing the window.

When you first create a button with a specific ID (in a given window), you must specify all the parameters up to and including `type`. If you later want to modify that button's characteristics, execute `Button` again with the same ID, and specify one or more of the other parameters (except `type`, which cannot be altered). The button will be redrawn using the new characteristics that you specified; any parameter that you don't specify will not be altered.

<code>id&</code>	a positive or negative integer whose absolute value is in the range 1 through 2147483647. The number you assign must be different from all other scroll bars or buttons in that window. Negative values build invisible buttons. Positive values build visible buttons.						
<code>state</code>	The state may be: <table border="0"> <tbody> <tr> <td><code>_grayBtn</code></td> <td>(0/disabled)</td> </tr> <tr> <td><code>_activebtn</code></td> <td>(1/default/active)</td> </tr> <tr> <td><code>_markedBtn</code></td> <td>(2/selected)</td> </tr> </tbody> </table>	<code>_grayBtn</code>	(0/disabled)	<code>_activebtn</code>	(1/default/active)	<code>_markedBtn</code>	(2/selected)
<code>_grayBtn</code>	(0/disabled)						
<code>_activebtn</code>	(1/default/active)						
<code>_markedBtn</code>	(2/selected)						
<code>value, min, max</code>	generally an integer value for the initial, minimum, and maximum values of a control						
<code>title\$</code>	a string expression. This parameter is not used to set the text of <code>Appearance Manager</code> buttons defined with <code>_kControlStaticTextProc</code> or <code>_kControlEditTextProc</code> . See <code>Def SetButtonTextString</code> for information on how this may be accomplished.						
<code>rect</code>	a rectangle in local window coordinates. You can express it in either of two forms: <table border="0"> <tbody> <tr> <td><code>(x1, y1) - (x2, y2)</code></td> <td>Two diagonally opposite corner points.</td> </tr> <tr> <td><code>@rectAddr&</code></td> <td>Long integer expression or <code>Pointer</code> variable which points to an 8-byte struct such as a <code>Rect</code> type.</td> </tr> </tbody> </table>	<code>(x1, y1) - (x2, y2)</code>	Two diagonally opposite corner points.	<code>@rectAddr&</code>	Long integer expression or <code>Pointer</code> variable which points to an 8-byte struct such as a <code>Rect</code> type.		
<code>(x1, y1) - (x2, y2)</code>	Two diagonally opposite corner points.						
<code>@rectAddr&</code>	Long integer expression or <code>Pointer</code> variable which points to an 8-byte struct such as a <code>Rect</code> type.						
<code>type</code>	any of the many types listed in the following text.						

New Things To Keep In Mind

On button creation, default values supplied for missing parameters (if any) are:

```
state      _activeBtn
value      1
min        0
max        1
title$     null string
```

You can hide the control with either `Button -1` or `Appearance Button -1` and you can deactivate the control with either `Button 1, _grayBtn` or `Appearance Button 1, _grayBtn`. Buttons are commonly hidden and revealed as tab panes are brought into or removed from view. The same is true of panes that are changed in response to items such as group pop-up placards.

To read an appearance button's value, use either:

```
x = Button(id)
or
x = Button(id, _FBGetCtlRawValue)
```

Summary of Appearance Helpers:

The following utility routines will help access information regarding the new appearance buttons:

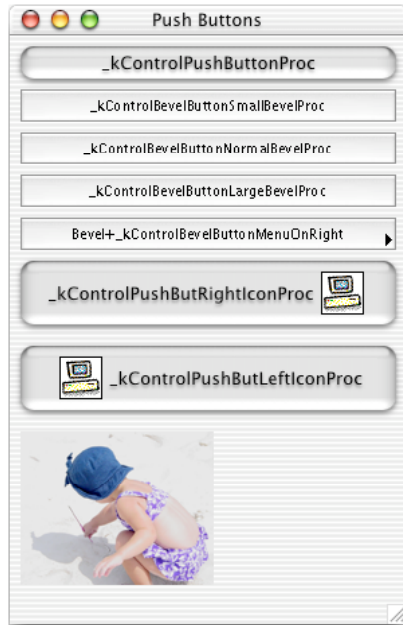
```
actualSize = Fn ButtonDataSize(btnID, part, tagName)
Def GetButtonData(btnID, part, tagName, maxSize, -
                  theData, actualSize)
Def SetButtonTextString(btnID, theString)
theString = Fn ButtonTextString$(btnID)
Def SetButtonFocus(btnID)
Def GetButtonTextSelection(btnID, selStart, selEnd)
Def SetButtonTextSelection(btnID, selStart, selEnd)
```

Button Types:

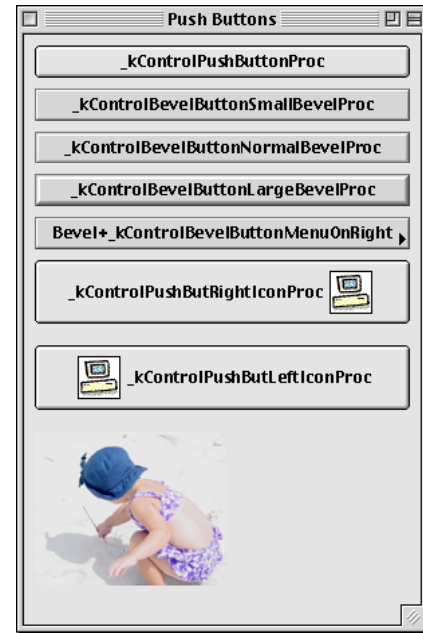
The Appearance Manager introduces many new control definitions. While it is not our intention to completely document all of Apple's new buttons, a few possible types are shown below with examples on how each might be implemented

Push Buttons:

Since the Appearance Manager works in both System 9 and OS-X, you will see major differences in how each control is presented. Common push buttons are shown below in both versions.



Push buttons in OS-X



Push buttons in System 9

The following code shows the source used to generate the displays.

```

/*      Appearance Button [#] [-] id[, [state][, [value][, -
      [min][, [max][, [title$][, [rect][, [type]]]]]] */
Appearance Button bRef, _activeBtn, 0, 0, 1, -
    "_kControlPushButtonProc", @r, _kControlPushButtonProc

Appearance Button bRef, _activeBtn, 0, 0, 1, -
    "_kControlBevelButtonSmallBevelProc", @r, -
    _kControlBevelButtonSmallBevelProc

Appearance Button bRef, _activeBtn, 0, 0, 1, -
    "_kControlBevelButtonNormalBevelProc", @r, -
    _kControlBevelButtonNormalBevelProc

Appearance Button bRef, _activeBtn, 0, 0, 1, -
    "_kControlBevelButtonLargeBevelProc", @r, -
    _kControlBevelButtonLargeBevelProc

// "value" is menu ID
Appearance Button bRef, _activeBtn, 101, 0, 1, -
    "Bevel+_kControlBevelButtonMenuOnRight", @r, -
    _kControlBevelButtonSmallBevelProc + -
    _kControlBevelButtonMenuOnRight

// max value is cicon ID
Appearance Button bRef, _activeBtn, 0, 0, 256, -
    "_kControlPushButRightIconProc", @r, -
    _kControlPushButRightIconProc

Appearance Button bRef, _activeBtn, 0, 0, 256, -
    "_kControlPushButLeftIconProc", @r, -
    _kControlPushButLeftIconProc

```

```

// get rect from pict to determine Button size
h = Fn GetPicture(256)
Long If h
    pR;8 = @h..picFrame%
    offsetrect(pR,-pR.left,-pR.Top)
    offsetrect(pR,r.left,r.Top)
// control "value" is pict ID
    Appearance Button bRef,_activeBtn,256,0,1,-
        "_kControlPictureProc",@pR,_kControlPictureProc
End If

```

Not all possible push buttons (and their variations) are shown here. For example, the control that displays an arrow to indicate the presence of a menu was built with a small bevel. It would have been created with a large bevel by using

```
_kControlBevelButtonLargeBevelProc + _kControlBevelButtonMenuOnRight
```

Other button types that you may wish to investigate are:

```

_kControlIconProc
_kControlIconNoTrackProc
_kControlIconSuiteProc
_kControlIconSuiteNoTrackProc
_kControlPictureNoTrackProc

```

Using Buttons to Group or Separate:

FB buttons (which are Control Manager controls) can be grouped together, placed in placards or separated by lines. The following example creates buttons on a plain white background so that you may more easily see the drawing that is implemented by the control definition. We begin with the source code statements used to create the buttons.

```

Appearance Button bRef,_activeBtn,0,0,1,-
    "_kControlGroupBoxTextTitleProc",@r,_kControlGroupBoxTextTitleProc

Appearance Button bRef,_activeBtn,0,0,1,-
    "_kControlGroupBoxSecondaryTextTitleProc",@r,-
    _kControlGroupBoxSecondaryTextTitleProc

Appearance Button bRef,_activeBtn,1,0,1,-
    "_kControlGroupBoxCheckBoxProc",@r,_kControlGroupBoxCheckBoxProc

Appearance Button bRef,_activeBtn,1,0,1,-
    "_kControlGroupBoxSecondaryCheckBoxProc",@r,-
    _kControlGroupBoxSecondaryCheckBoxProc

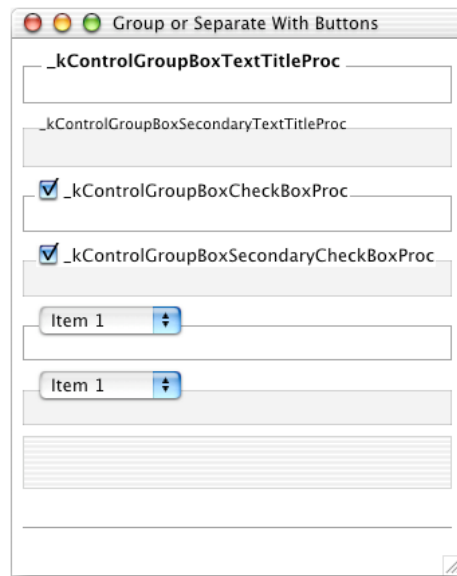
// min value is menu ID
Appearance Button bRef,_activeBtn,1,101,1,-
    "",@r,_kControlGroupBoxPopUpButtonProc

Appearance Button bRef,_activeBtn,1,101,1,-
    "",@r,_kControlGroupBoxSecondaryPopUpButtonProc

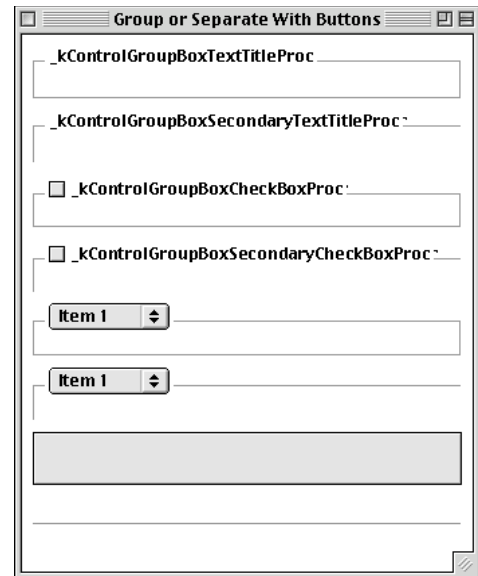
Appearance Button bRef,_activeBtn,1,0,1,"",@r,_kControlPlacardProc

Appearance Button bRef,_activeBtn,1,0,1,-
    "",@r,_kControlSeparatorLineProc

```



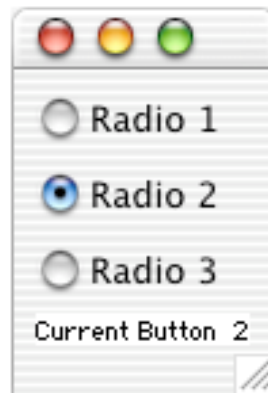
Groups & Separators in OS-X



Groups and Separators in System 9

Embedding Buttons:

Part of the strength of Appearance Manager buttons is that one button may be embedded in another. By disabling or hiding the parent button (called a super control), all embedded controls would automatically be disabled or hidden. Each window has a primary control known as a root control. The following example builds a window with a parent radio group button. Inside of that parent are three radio buttons. We can determine which of the three buttons has been selected by getting the value (via the `Button()` function) of the parent button.



FUTUREBASIC REFERENCE

```

Dim r      As Rect
Dim pR     As Rect
Dim h      As Handle
Dim bRef   As Long
Dim err    As OSerr

// setup
_btnHt      = 20
_btnWd      = 80
_btnMargin  = 8
bRef        = 1

// create a Window
SetRect(r,0,0,_btnWd_btnMargin_btnMargin,120)

Appearance Window 1,,@r

err = Fn SetThemeWindowBackground (Window( _wndPointer ),-,
    _kThemeActiveDialogBackgroundBrush, _zTrue )

// Button #1 is the papa button
// note that the parent button has sufficient space so that
// it holds all embedded buttons within its own rectangle

SetRect(r,_btnMargin,_btnMargin,-
    _btnMargin_btnWd,(_btnMargin_btnHt)*3)
Appearance Button bRef,_activeBtn,0,0,1,"",@r,_kControlRadioGroupProc

bRef ++
SetRect(r,_btnMargin,_btnMargin,_btnMargin_btnWd,_btnMargin_btnHt)
Appearance Button bRef,_activeBtn,0,0,1,-
    "Radio 1",@r,_kControlRadioButtonProc
Def EmbedButton(bRef,1)

bRef ++ : OffsetRect(r,0,_btnHt_btnMargin)
Appearance Button bRef,_activeBtn,0,0,1,-
    "Radio 2",@r,_kControlRadioButtonProc
Def EmbedButton(bRef,1)

bRef ++ : OffsetRect(r,0,_btnHt_btnMargin)
Appearance Button bRef,_activeBtn,0,0,1,-
    "Radio 3",@r,_kControlRadioButtonProc
Def EmbedButton(bRef,1)

Local Fn HandleDialog
    Dim As Long action,reference
    action    = Dialog(0)
    reference = Dialog(action)
    Long If action = _btnClick
        MoveTo(8,100)
        Print "Current Button ";Button(1);
    End If
End Fn

On Dialog Fn HandleDialog

Do
    HandleEvents
Until 0

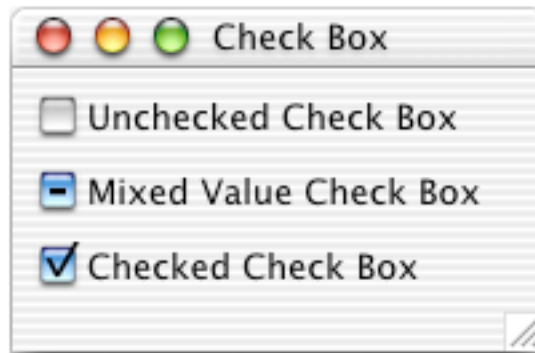
```

Check Boxes

Other than the obvious differences in physical appearance, check boxes generally follow the same guidelines as they have for many years. One notable exception to this rule is the ability to create a mixed check box. This box contains a dash instead of a check mark to show that part, but not all, of the current selection has a specific feature. This adds a new possible maximum value of 2 (`_kControlCheckBoxMixedValue = 2`) to the control's range.

Possible check box values now include:

```
_kControlCheckBoxUncheckedValue
_kControlCheckBoxCheckedValue
_kControlCheckBoxMixedValue
```



Check Boxes

The buttons in the screen shot above were created using the following lines of code:

```
Appearance Button bRef,_activeBtn,~
    _kControlCheckBoxUncheckedValue,0,~
    _kControlCheckBoxMixedValue,~
    "Unchecked Check Box",@r,_kControlCheckBoxProc

Appearance Button bRef,_activeBtn,~
    _kControlCheckBoxMixedValue,0,~
    _kControlCheckBoxMixedValue,~
    "Mixed Value Check Box",@r,_kControlCheckBoxProc

Appearance Button bRef,_activeBtn,~
    _kControlCheckBoxCheckedValue,0,~
    _kControlCheckBoxMixedValue,~
    "Checked Check Box",@r,_kControlCheckBoxProc
```

Note:

You cannot use `Button bRef,state` to tick and untick group buttons of type `_kControlGroupBoxCheckBoxProc` and `_kControlGroupBoxSecondaryCheckBoxProc`. Use `Appearance Button bRef,,state-1` instead. (`Button bRef,0` and `Button bRef,1` will however inactivate and activate the button respectively).

Time and Date Buttons:

In addition to more common controls, the Appearance Manager can create buttons that manage dates and times. Special data structures are maintained to access the information from these controls, but by following a few simple examples, you can quickly master these skills.

The enhanced `Button` function is useful for extracting complex data from controls. Two specific items come in to play:

```
ignored = Button(btnRef, _FBGetControlDate)
ignored = Button(btnRef, _FBGetControlTime)
```

Referencing either of these functions will fill a global date/time record named `gFBControlLongDate` and another named `gFBControlSeconds`. The variable named `gFBControlSeconds` is a signed 64 bit variable which may be saved in a file or used in any variable where compressed storage is required.

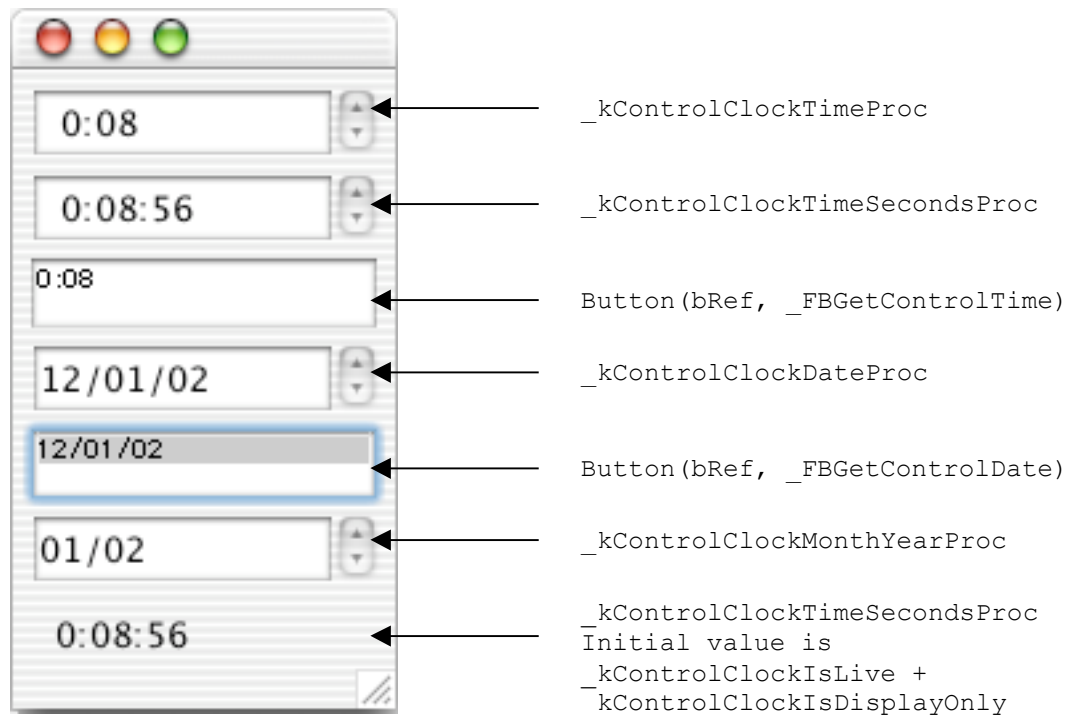
The format for `gFBControlLongDate` is that of a `LongDateRec` which follows the layout of the structure below:

```
Begin Record LongDateRec
  Dim era           As Short
  Dim year          As Short
  Dim month         As Short
  Dim day           As Short
  Dim hour          As Short
  Dim minute        As Short
  Dim second        As Short
  Dim dayOfWeek     As Short
  Dim dayOfYear     As Short
  Dim weekOfYear    As Short
  Dim pm            As Short
  Dim res1          As Short
  Dim res2          As Short
  Dim res3          As Short
End Record
```

After calling the `Button` function to examine the contents of a control, you may extract portions of the date/time as follows:

```
dayOfMonth = gFBControlLongDate.day
thisYear   = gFBControlLongDate.year
```

Another variable is maintained that holds the text for a specific date/time control. The contents of `gFBControlText` (a Pascal string) are determined by the second `Button` function parameter. When `_FBGetControlDate` is used, it is the control's date. When `_FBGetControlTime` is used, it is the control's time.

*Date/Time Buttons*

In addition to setting specific types when creating a date/time control, you must set an initial value of one of the following:

```
_kControlClockNoFlags
_kControlClockIsDisplayOnly
_kControlClockIsLive
```

The specific statements used to create the Time/Date example follow:

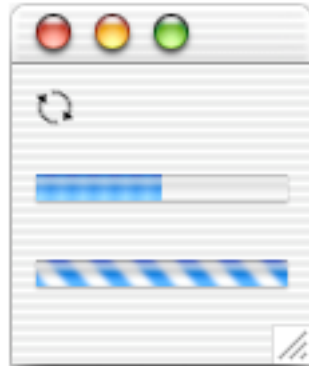
```
Appearance Button bRef,_activeBtn,0,0,1,,@r,¬
    _kControlClockTimeProc
Appearance Button bRef,_activeBtn,_kControlClockIsLive,0,1,,¬
    @r,_kControlClockTimeSecondsProc
Appearance Button bRef,_activeBtn,_kControlClockNoFlags,¬
    0,1,,@r,_kControlClockDateProc
Appearance Button bRef,_activeBtn,_kControlClockNoFlags,¬
    0,1,,@r,_kControlClockMonthYearProc
Appearance Button bRef,_activeBtn,¬
    _kControlClockIsLive_kControlClockIsDisplayOnly,¬
    0,1,,@r,_kControlClockTimeSecondsProc
```

To extract and display the contents of a control, the following statements were created:

```
err = Button (bRef,_FBgetControlTime)
Edit Field bRef,gFBControlText,@r
```

Wait States:

The Appearance Manager provides several methods for telling the user that your application is busy with a task. These include chasing arrows, and finite and indeterminate progress bars.



Wait States

The chasing arrows control is easy to create and is self maintaining. Each time your program scans for events, the arrows are animated. The following statement creates a chasing arrows control:

```
Appearance Button bRef,_activeBtn,0,0,1,,@r,_kControlChasingArrowsProc
```

Progress bars are also easy to create, but you need to keep a couple of things in mind. First, the progress bar operates in a range of -32,768 to +32,767. If your task involves a greater number of steps, you will have to calculate a ratio to keep things within range. Second, the progress bar is updated by your program. This is as easy as setting a new value for the button, but it is code that you must write.

The minimum and maximum values for the control become the minimum and maximum values for the progress bar. The initial and current value show the current rate of progress. In the example above, the button was created using the following source:

```
Appearance Button bRef,_activeBtn,50,0,100,,@r,_  
_kControlProgressBarProc
```

The minimum value was zero; maximum was 100. At the time of creation, the control value was 50, so the indicator shows colorization half way across the bar. If we wanted to indicate that the next step had been completed, we would use the following code:

```
Appearance Button bRef,,51
```

Indeterminate progress bars are more complex. After the button is created, you must set the control's internal data to a new value. The following code shows how:

```
Appearance Button bRef,_activeBtn,1,0,1,,@r,_kControlProgressBarProc  
  
Dim b As Boolean  
Dim err As oSErr  
  
b = _True  
  
err = Fn SetControlData(bRef, 0,_  
_kControlProgressBarIndeterminateTag,SizeOf(Boolean), @b)
```

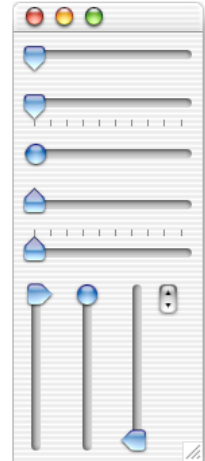

Range Selectors (Sliders and Arrows):

There are many variations of the slider. Each begins with the simple type constant of `_kControlSliderProc`. Additional parameters are added to this constant to add features to the control. The following constants are available for slider variations:

```
_kControlSliderLiveFeedback
_kControlSliderHasTickMarks
_kControlSliderReverseDirection
_kControlSliderNonDirectional
```

To create a slider that uses an upward pointing indicator and has tickmarks, the following type would be used:

```
_kControlSliderProc + _kControlSliderHasTickMarks + ¬
_kControlSliderReverseDirection
```



*Range Selectors,
sliders & little
arrows*

Vertical sliders are created by building the button with a vertical dimension that is greater than the horizontal dimension. The control definition handles the new orientation automatically.

The following source lines show how this display was created:

```
Appearance Button bRef,_activeBtn,1,1,10,,@r,_kControlSliderProc
Appearance Button bRef,_activeBtn,10,1,10,,@r,¬
_kControlSliderProc_kControlSliderHasTickMarks
Appearance Button bRef,_activeBtn,1,1,10,,@r,¬
_kControlSliderProc_kControlSliderNondirectional
Appearance Button bRef,_activeBtn,1,1,10,,@r,¬
_kControlSliderProc_kControlSliderReverseDirection
Appearance Button bRef,_activeBtn,10,1,10,,@r,¬
_kControlSliderProc_kControlSliderHasTickMarks + ¬
_kControlSliderReverseDirection
// vert orientation
Appearance Button bRef,_activeBtn,10,1,10,,@r,_kControlSliderProc
Appearance Button bRef,_activeBtn,10,1,10,,@r,¬
_kControlSliderProc_kControlSliderNondirectional
Appearance Button bRef,_activeBtn,10,1,10,,@r,¬
_kControlSliderProc_kControlSliderHasTickMarks + ¬
_kControlSliderReverseDirection
Appearance Button bRef,_activeBtn,0,0,1,,@r, ¬
_kControlLittleArrowsProc
```

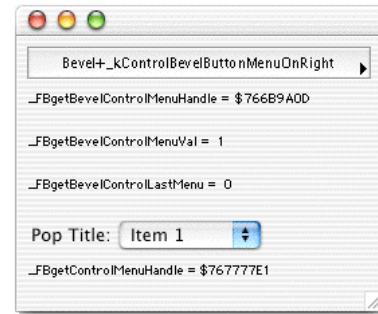
When sliders are created, the number of tick marks is set by the initial value of the control. After the control is created, the value is reset to the control minimum. Sliders range from a minimum value of -32,768 to a maximum of +32,767. The number of tick marks is something that you need to determine by balancing the size of the control against the range of the control's minimum/maximum value.

The little arrows (shown in the screen shot above) are used to increment and decrement a related visual counter (usually an edit field with a specific range of numbers). The current version of the OS-X control definition is intolerant of variations in the value used for the height of this type of control. Our tests show that it must be exactly 22 pixels tall. Other values offset the arrows inside of the beveled area or, in more extreme cases, can place the arrows entirely outside of the beveled area.

Pop-Up Menus:

There are two distinct types of pop-up menus: beveled, and standard. Both are valid types and the particular use of one over the other is something that should be guided by your individual application and by Apple's Human Interface Guidelines. Beveled buttons are created as follows:

```
Appearance Button bRef, _activeBtn, menuID, 0, 1, -1,
    "Bevel+ _kControlBevelButtonMenuOnRight", -1,
    @r, _kControlBevelButtonSmallBevelProc + -1,
    _kControlBevelButtonMenuOnRight
```



Pop-Up Menu Buttons

When bevel-button menus are created, the initial value for the control is the resource ID number of the menu. Three specific `Button` function commands may be used to extract information from the control.

```
Handle      = Button(bRef, _FBgetBevelControlMenuHandle)
currentItem = Button(bRef, _FBgetBevelControlMenuVal)
previousMenu = Button(bRef, _FBgetBevelControlLastMenu)
```

Standard pop-up buttons follow slightly different syntax. When creating, the minimum value specifies the menu resource ID and the maximum value is the width of the title for the menu. Passing in a menu ID of -12345 causes the popup not to try and get the menu from a resource. Instead, you can build the menu and later stuff the menuhandle field in the popup data information (using `Def SetButtonData(id&, _kControlMenuPart, -1, _kControlPopupButtonMenuHandleTag, SizeOf(Handle), @yourMenuHndl)`). You can pass -1 in the *max* parameter to have the control calculate the width of the title on its own instead of guessing and then tweaking to get it right. It adds the appropriate amount of space between the title and the popup. A maximum value of zero means, "Don't show the title."

After creation you might need to change the value, minimum and maximum to the correct settings for your pop-up menu with:

```
Appearance Button id&, ,value,min,max
```

The standard pop-up button menu in the above illustration was created with the following code:

```
Appearance Button bRef, _activeBtn, 0, 101, -1, "Pop Title:" -1,
    , @r, _kControlPopupButtonProc
```

A single `Button` function provides access to the menu handle. Remember: standard and beveled pop-up menus do not use the same `Button` function constants.

```
menuHandle = Button(bRef, _FBgetControlMenuHandle)
```

You can retrieve the current pop-up menu item with:

```
mItem = Button(bRef)
```

List Boxes:

Lists generally use an auxiliary resource to define their format. The resource type used is 'ldes' and a definition for it can be created using Resorcerer 2.4 or later or by creating a template in any resource editor. You may also format a handle to match the `_"ldes"` record and save that handle as a resource. The resource ID for the ldes is passed in the *value* parameter when creating the control. You may pass zero in *value* which would tell the List Box control to not use a resource. The list will be created with default values, and will use the standard LDEF (0). You can change the list by getting the list handle. You can set the LDEF by using the tag below (`_kControlListBoxLDEFTag`) in conjunction with `Def SetButtonData`.

A list box resource is defined as follows:

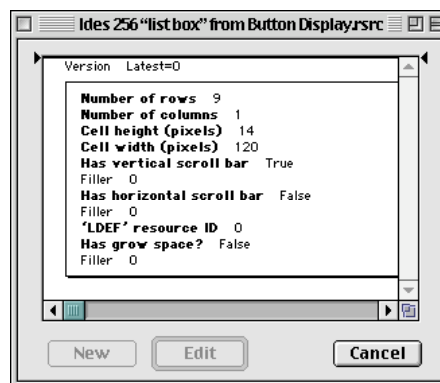
```

Begin Record ldes
  Dim versionNumber      As Short
  Dim numberOfRows      As Short
  Dim numberOfColumns    As Short
  Dim cellHeight         As Short
  Dim cellWidth          As Short
  Dim hasVertScroll      As Boolean
  Dim filler1            As Byte
  Dim hasHorizScroll     As Boolean
  Dim filler2            As Byte
  Dim LDEFresID          As Short
  Dim hasSizeBox         As Boolean
  Dim reserved          As Byte
End Record

```

The following example creates a list box from a resource with an ID of 256. Then it fills the list with item text.

To simplify this example, Resorcerer was used to create the list box resource.



Creating an ldes Resource

```

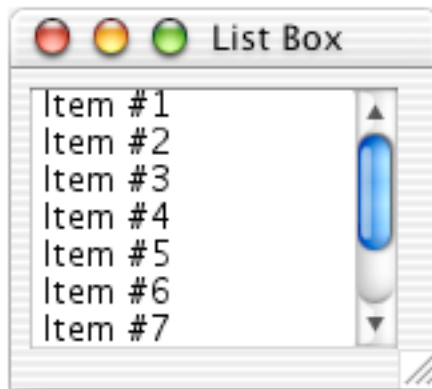
Appearance Button bRef, _activeBtn, 256, 0, 1, -
    "List Box", @r, _kControlListBoxProc

Dim cH          As Handle //control handle
Dim @bufferSize As Long
Dim @lH          As Handle //list handle
Dim y, t$
Dim @celly, cellX //cell "point" Record

cH = Button&(bRef)
err = Fn GetControlData( cH, 0, _kControlListBoxListHandleTag, -
    SizeOf(Handle), lH, bufferSize )

cellX = 0
For celly = 0 To 9
    t$ = "Item #" + Mid$(Str$(celly+1), 2)
    LSetCell(@t$(1), t$(0), celly, lH)
Next

```



List Box

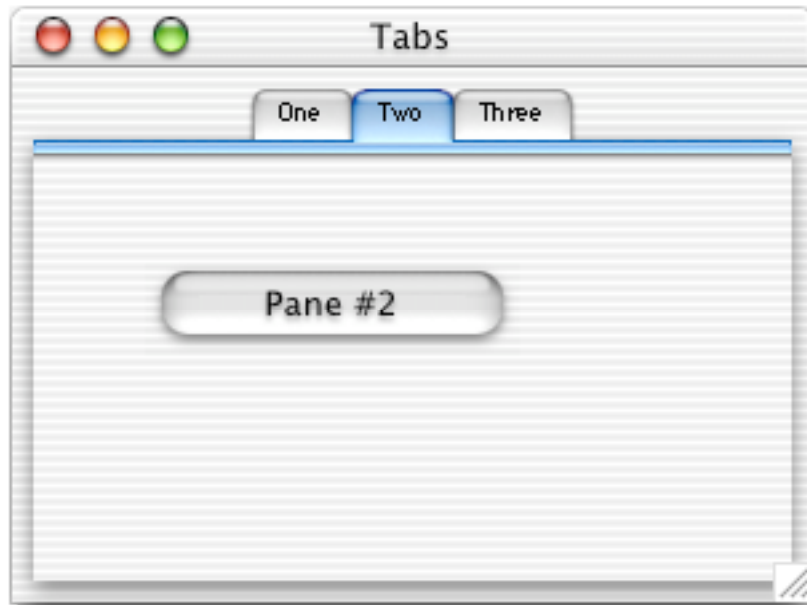
Tab Buttons:

Tab buttons will require more work than other controls. This stems from the fact that tabs are really several controls that act in unison. First is the main tab control. When this is created you specify the number of tabs that will be present by setting the max value of the control. It is generally better to create the tab button invisibly (by using a negative button reference number) then show it by issuing a `Button` statement with the positive version of the reference number.

After the initial shell is built for the tab, you must set the title for each tab using `Def SetButtonData`. Then a user pane is inserted for each tab. These are embedded in the tab shell using `Def EmbedButton`. Buttons that will reside in each user pane are created and embedded in the user pane.

When a dialog event is encountered, the value of the tab shell button corresponds to the position of the clicked tab in the tab list. Your program must loop through each user pane and show or hide them so that the display matches the clicked tab.

Study the following example to see how a working tab button is created. Be sure to note the simple `Dialog` handler that maintains the buttons.



Tab Buttons

There are many styles of tab buttons:

```
_kControlTabLargeProc  
_kControlTabsmallProc  
_kControlTabLargeNorthProc  
_kControlTabsmallNorthProc  
_kControlTabLargeSouthProc  
_kControlTabsmallSouthProc  
_kControlTabLargeEastProc  
_kControlTabsmallEastProc  
_kControlTabLargeWestProc  
_kControlTabsmallWestProc
```

FUTUREBASIC REFERENCE

This example uses `_kControlTAbSmallProc`, but you should experiment with other types to see the results.

```
Dim r      As Rect
Dim x      As Long
Dim bRef   As Long
Dim infoRec As ControlTabInfoRec

// Names of the individual tabs

_tabCount = 3
Dim tabTitles$( _tabCount )
tabTitles$(1) = "One"
tabTitles$(2) = "Two"
tabTitles$(3) = "Three"

// create a Window

SetRect( r, 0, 0, 300, 200 )
Appearance Window 1, "Tabs", @r, _kDocumentWindowClass

Def SetWindowBackground( _kThemeActiveDialogBackgroundBrush, _zTrue )
/*

    Button 100 is the tab 'shell'. In this example, it is made
    to be the full size of the Window, less a small margin.
    buttons 1, 2, & 3 will be the embedded user panes that
    contain information to be displayed for each tab.

    A tab control is usually built as invisible. This is because
    the information contained in the tabs will be modified as the
    control is being constructed. Making it visible after all
    modifications have been completed provides a cleaner Window
    build.

*/

_tabBtnRef = 100
_btnMargin = 8
InsetRect( r, _btnMargin, _btnMargin )
Appearance Button -_tabBtnRef, 0, 0, 2, _tabCount,, @r,-
    _kControlTAbSmallNorthProc

/*

    Fix the tab to use a small font. This is not a requirement,
    but it is information which many will find useful.

*/

Dim cfsRec As ControlFontStyleRec
cfsRec.flags = _kControlUseSizeMask
cfsRec.size = 9

Def SetButtonFontStyle( _tabBtnRef, cfsRec )
/*

    Adapt a rectangle that can be used for the content area of each tab.

*/

InsetRect( r, _btnMargin, _btnMargin )
r.Top += 20
```

```

// Loop thru the tabs and set up individual panes

For x = 1 To _tabCount
    infoRec.version      = _kControlTabInfoVersionZero
    infoRec.iconSuiteID = 0
    infoRec.Name         = tabTitles$(x)
    Def SetButtonData( _tabBtnRef, x, _kControlTabInfoTag, _
        SizeOf( infoRec ), infoRec )

    /*
        Each of these panes is a button that is embedded in the tab
        button. The first one will be visible. All others will be
        invisible because information from only one tab at a time
        can be viewed.

        Remember: negative button reference numbers make invisible
        buttons.

        Once a new pane button (_kControlUserPaneProc) is created,
        it is embedded into the larger tab button.
    */

    If x != 1 Then bRef = -x Else bRef = x
    Appearance Button bRef,,, _
        _kControlSupportsEmbedding,,, @r, _kControlUserPaneProc
    Def EmbedButton( x, _tabBtnRef )
Next

/*
    Now we have a tab shell (_tabBtnRef = 100) and in it we have
    embedded three tab panes (1,2, and 3). To demonstrate how
    these can contain separate info, we will place a simple
    button in each of the three panes.

    Button 10 in pane 1
    Button 20 in pane 2
    Button 30 in pane 3
*/

InsetRect( r, 32, 32 )
r.botTom = r.Top + 24
r.right  = r.left + 128

Appearance Button 10, _activeBtn,,,, _
    "Pane #1", @r, _kControlPushButtonProc
Def EmbedButton( 10, 1 )

OffsetRect( r, 8, 8 )
Appearance Button 20, _activeBtn,,,, _
    "Pane #2", @r, _kControlPushButtonProc
Def EmbedButton( 20, 2 )

OffsetRect( r, 8, 8 )
Appearance Button 30, _activeBtn,,,, _
    "Pane #3", @r, _kControlPushButtonProc
Def EmbedButton( 30, 3 )

Button _tabBtnRef, 1 // make visible

```

FUTUREBASIC REFERENCE

```
/*
    Only one event (a button click in the tab shell button) gets
    a response from our Dialog routine. The value returned (1,2,
    or 3) corresponds to buttons 1,2, or 3 that were embedded
    into the tab parent.

    Our only action is to show (Button j) or hide (Button -j) the
    proper tab pane. All controls embedded in those panes will
    automatically be shown or hidden.
*/

Local Fn doDialog
    Dim As Long action, reference, j
    action = Dialog(0)
    reference = Dialog(action)
    Long If action == _btnClick And reference == _tabBtnRef
        For j = 1 To _tabCount
            Long If j == Button(_tabBtnRef)
                Button j
            Xelse
                Button -j
            End If
        Next
    End If
End Fn

On Dialog Fn doDialog
Do
    HandleEvents
Until 0
```

See Also:

Button& function; Button function; Button Close; Scroll Button;
Dialog function; Def EmbedButton; Def SetButtonData

Appearance Window

Statement

✓ *Appearance*

✗ *Standard*

✗ *Console*

Syntax:

```
Appearance Window [#] [-] id&[, [title$][, [rect][, [WindowClass][, ~
[WindowAttributes] [, [FBAttributes]]]]]
```

Revision:

February, 2002 (Release 6)

Description:

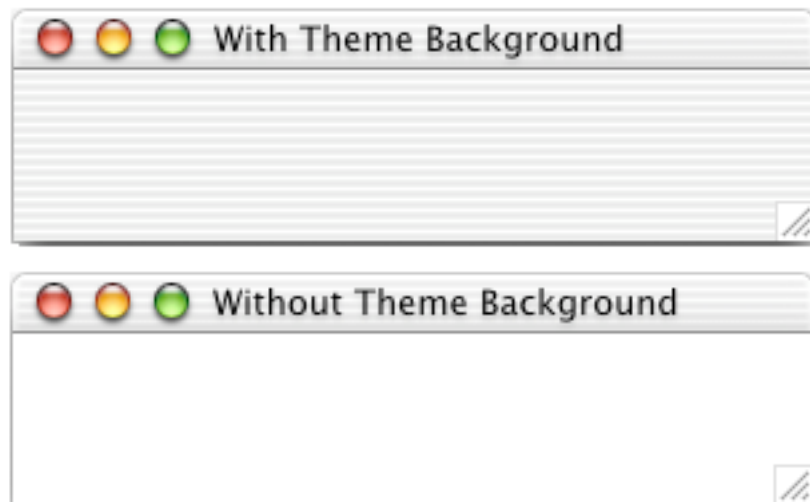
Use this statement to do any of the following:

- Create a new screen window;
- Activate (highlight and bring to the front) an existing window;
- Make an existing window visible or invisible;
- Alter the title or rectangle of an existing window.

The `Appearance Window` statement closely follows the older `Window` statement, but is used primarily for the creation of windows. You may freely mix `Window` functions and `Window` statements with windows that are created via the `Appearance Window` statement. For instance, after creating a window with the `Appearance Window` statement, you could determine its size with the `Window(_width)` and `Window(_height)` functions.

You may notice that windows in most modern applications have a background that is something other than white. This is not accomplished by drawing into the window with graphic commands. It is put in place with a simple call to the Theme Manager. The following line will be useful in many programs.

```
Def SetWindowBackground(_kThemeActiveDialogBackgroundBrush,_zTrue)
```



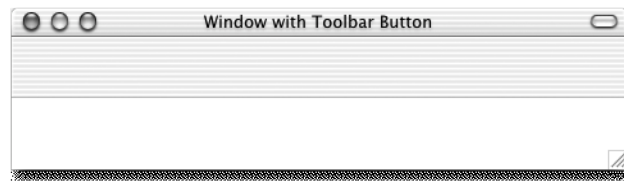
Windows with and without theme backgrounds

The parameters for `Appearance Window` should be specified as follows. They are interpreted slightly differently depending on whether you are creating a new window or altering an existing one.

- `id` a positive or negative integer whose absolute value is in the range 1 through 2147483647.
- `title$` a string expression.
- `rect` a rectangle in global screen coordinates. You can express it in either of two forms:
 - `(x1, y1) - (x2, y2)` Two diagonally opposite corner points.
 - `@rectAddr&` Long integer expression or `Pointer` variable which points to an 8-byte struct such as a `Rect` type.
- `WindowClass` an unsigned long integer that specifies the Macintosh window class. This is not the same value as FB's user definable class for the standard runtime. It more closely represents the layer in which a window will reside. To create a `windowClass` variable use the following syntax:

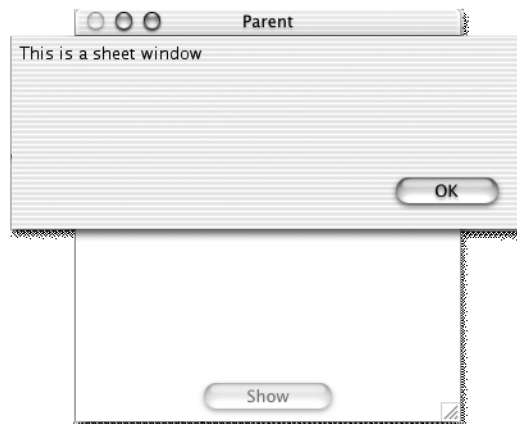
```
Dim wc As WindowClass
```

The `windowClass` table introduces some new terms which may not be familiar to those new to OS-X.



Window with toolbar

A sheet window is attached to a parent window. It drops from the title bar of the parent and is used to force some decision relative to the parent window.



Sheet window

<i>WindowClass</i>	<i>Value</i>	<i>Description</i>
<code>_kAlertWindowClass</code>	1	I need your attention now.
<code>_kMovableAlertWindowClass</code>	2	I need your attention now, but I'm kind enough to let you switch out of this app to do other things.
<code>_kModalWindowClass</code>	3	system modal, not draggable
<code>_kMovableModalWindowClass</code>	4	application modal, draggable
<code>_kFloatingWindowClass</code>	5	floats above all other application windows Available in OS 8.6 or later
<code>_kDocumentWindowClass</code>	6	document windows
<code>_kDeskTopWindowClass</code>	7	the desktop
<code>_kHelpWindowClass</code>	10	help windows
<code>_kSheetWindowClass</code>	11	sheets
<code>_kToolbarWindowClass</code>	12	floats above docs, below floating windows
<code>_kPlainWindowClass</code>	13	plain
<code>_kOverlayWindowClass</code>	14	overlays
<code>_kSheetAlertWindowClass</code>	15	sheet alerts
<code>_kAltPlainWindowClass</code>	16	plain alerts

- *WindowAttributes* – this unsigned long integer describes the features and widgets available to a window such as a close box, grow box, or a collapse box. You can dimension a *windowAttributes* variable as follows:

Dim wa **As** WindowAttributes

The appearance of window widgets will vary from one version of the system software to the next. For instance, in System 9, the vertical, horizontal and full zoom boxes have distinct representations. In OS-X there is no visible difference.



Vertical, horizontal, and full zoom in System 9



Zoom in OS-X

WindowAttribute	Value	Description
_kWindowNoAttributes	0	none
_kWindowCloseBoxAttribute	1	close box
_kWindowHorizontalZoomAttribute	2	horizontal zoom
_kWindowVerticalZoomAttribute	4	vertical zoom
_kWindowFullZoomAttribute	6	standard zoom
_kWindowCollapseBoxAttribute	8	collapse box (sends to OS-X dock)
_kWindowResizableAttribute	16	grow box
_kWindowSideTitlebarAttribute	32	title on side for floating window
_kWindowNoUpdatesAttribute	65536	does not receive update event
_kWindowActivatesAttribute	131072	does not receive activate event
_kWindowToolbarButtonAttribute	64	has a toolbar button in title bar
_kWindowNoShadowAttribute	2097152	no drop shadow
_kWindowLiveResizeAttribute	268435456	resize events repeatedly sent while window is being sized
_kWindowStandardDocumentAttributes	31	close box, zoom box, collapse box, grow box
_kWindowStandardFloatingAttributes	9	close box, collapse box

- *FBAttributes* – this long integer sets up handling procedures that are used by the runtime. Some of the features used in the Standard BASIC runtime are not carried forward for the Appearance Runtime. For example, `_noAutoClip` is not used because there is no such thing as `AutoClip` in Appearance.

<i>constant</i>	<i>Value</i>	<i>FB Attribute</i>
<code>_updateVisRgn</code>	2048	This attribute affects how the window's clip region will be set when FutureBASIC3 calls your dialog-event handling routine with a <code>_wndRefresh</code> event. If you specify this attribute, the clip region will be set to include only that part of the window which was identified as actually needing a refresh (the clip region will be reset to its previous value when the routine exits). If you omit this attribute, the clip region will be set to include the entire window (possibly excluding controls, edit fields, etc.)
<code>_clickThru</code>	4096	This attribute affects what happens when your program activates the window in response to a <code>_wndClick</code> event. If the <code>_clickThru</code> attribute is set, the activating click will be "passed through" to the window; this may cause other events (such as <code>_btnClick</code> or <code>_efClick</code>) to be generated, depending on what was clicked on. If you omit this attribute, two separate clicks are required to activate the window and to interact with its contents.
<code>_noAutoFocus</code>	32768	Use this attribute to prevent the tab key from advancing the keyboard focus in edit fields
<code>_keepInactive</code>	65536	This attribute insures that the window will never be activated. Controls in the window will not function. If you bring the window forward under program control (<code>Window</code> statement) it will behave normally. This type of window is intended for use as a backdrop.

To Create a New Window

- Specify an id value such that `Abs(id)` is different from the ID number of any existing window. A new window is created and is assigned an ID number of `Abs(id)`. You can use the window's ID number later to identify the window in other FB^3 statements and functions. If id is negative, the window is created invisibly; it's sometimes useful to create a window invisibly if it will contain controls, edit fields and graphics that may take a long time to build. You can use the `Window` statement again to make an invisible window visible (see below). When you create a new window, it becomes the current output window. If you create it visibly (and you don't specify the `_keepInBack` attribute), it also becomes the current active window.
- `title$` assigns a string to the window's title bar (if the window has a title bar). If you omit this parameter, the window will be created without a title.
- `rect` specifies the initial size and location of the window's content rectangle. Note that `rect` does not include the window's frame. This parameter is interpreted in a special way if you specify an upper-left coordinate of (0,0) in `rect`; in this case, the window is centered in the screen, and its width and height are determined by the right and bottom coordinates of `rect`. Note that this special interpretation applies only when you're creating a new window. If you omit this parameter, a window of a "default" size and location is created.
- `WindowClass` specifies the layer in which a window will reside.
- `WindowAttributes` specifies the types of window widgets (close box, grow box) that a window will include.
- `FBAttributes` specifies runtime handling parameters that determine how the window will behave.

See Also:

```
Def TransitionRect; Def WindowCategory; MinWindow; MaxWindow; SetZoom;
Get Window; Window Close; Window Output; Window function; Dialog function
```

Append	statement	
✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>

Syntax:

```
Append [#] fileID
```

Description:

This statement moves the file mark, in the currently-open file indicated by *fileID*, to the end-of-file position (without overwriting any of the existing data in the file). This causes subsequent file output statements such as `Print#`, `Write#` and `Write File#` to append data to the end of the file.

Example:

In the following, note that we open the output file using the “R” method. This is because opening with the “O” method causes the “end-of-file” mark to move to the beginning of the file, effectively erasing any existing data.

```
A$ = "TESTING..."
Open "R",1,fileName$,,wdRefNum%      'Open an existing file
Append #1                            'Set file pointer to end
Write #1, A$;25                       'Add data to tnd of file
Close #1                             'Close file
```

See Also:

```
Files$; Open; Close; Read#
```

Apple Menu	statement	
✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>

Syntax:

```
Apple Menu string$
```

Description:

This statement inserts one or more items at the top of the Apple Menu, and separates them from the existing Apple Menu items with a grey dividing line. The items you add are visible only while your application is frontmost. The *string*\$ parameter contains the text of the item(s); to add multiple items, separate them with semicolons in *string*\$. Certain “meta characters” in *string*\$ have special interpretations; see the `Menu` statement for more information.

After you add items to the Apple Menu, you can use the `Menu` function to detect when the user selects one of the items you’ve added.

If you execute `Apple Menu` more than once, any items you added to the Apple Menu previously will be completely replaced.

Example:

```
Apple Menu "About this program...;About Me..."
```



See Also:

```
Menu function; Menu statement; On Menu Fn; HandleEvents
```


AppleEventMessage\$ function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
string$ = AppleEventMessage$
```

Revision:

May, 2001 (Release 5)

Description:

When your application (also referred to as your process) receives an Apple Event message and the event class is `_"TEXT"`, FB will retrieve the event data as a Pascal string when this function is invoked.

Example:



CD Example: AppleEvents folder

See Also:

`SendAppleEvent`, `HandleEvents`, `On AppleEvent`, `GetProcessInfo`,
`Kill AppleEvent`

Asc**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

ASCIIcode% = Asc(string$)
ASCIIcode% = Asc(container$$)

```

Revision:

July 28, 2000 (Release 3)

Description:

Returns the ASCII character code for the first character in *string\$* or *container\$\$*. (A character code is a numeric value in the range of 0 through 255 that represents a specific character in the American Standard Code for Information Interchange (ASCII).) If *string\$* (or *container\$\$*) contains no characters, then *Asc(string\$)* returns zero.

The ASCII character codes between 32 and 127 represent standard characters which generally remain the same from one font family to another. Codes greater than 127 represent different sets of characters depending on the font. Codes below 32 usually represent non-printing “characters.”

Example:

CD Example: Asc.bas

Note:

If the string is a single-character literal, such as “G”, then you should consider using the underscore-literal syntax instead, as in this example:

```
ASCIIcode% = _"G"
```

The above code executes much faster than `ASCIIcode% = Asc("G")`.

You can use the following syntax to return the ASCII code of the *n*-th character in a string variable *stringVar\$*:

```
ASCIIcode% = stringVar$[n]
```

See Also:

`Chr$`; Appendix F: *ASCII Character Codes*

Asin**function**

✓ *Appearance***✓** *Standard***✓** *Console*

Syntax:

```
radianAngle# = Asin(expr)
```

Description:

Returns the arcsine of *expr* in radians. In other words, if *expr* represents the sine of some angle, then `Asin(expr)` returns the angle. The returned angle will be in the range of $-\pi/2$ to $+\pi/2$ radians (which corresponds to -90 to $+90$ degrees). `Asin` always returns a double-precision result.

See Also:

`Sin`; `Cos`; `Tan`; `Atn`; `Acos`

Asinh function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
result# = Asinh(expr)
```

Description:

Returns the inverse hyperbolic sine of *expr*. This is the inverse of the `Sinh` function, so that `Asinh(Sinh(x))` equals *x*. `Asinh` always returns a double-precision result.

See Also:

`Sinh`; `Acosh`; `Atanh`

Atan function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

radianAngle# = **Atan**(*expr*)

Description:

Atan is a synonym for the Atn function.

See Also:

Atn

Atanh function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

`result# = Atanh(expr)`

Description:

Returns the inverse hyperbolic tangent of *expr*. This is the inverse of the `Tanh` function, so that `Atanh(Tanh(x))` equals *x*. `Atanh` always returns a double-precision result.

See Also:

`Tanh`; `Acosh`; `Asinh`

Atn**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
radianAngle# = Atn(expr)
```

Description:

Returns the arctangent of *expr* in radians. In other words, if *expr* represents the tangent of some angle, then `Atn(expr)` returns the angle. The returned angle will be in the range of $-\pi/2$ to $+\pi/2$ radians (which corresponds to -90 to $+90$ degrees). `Atn` always returns a double-precision result.

Example:

Because `Atn(1)` equals $\pi/4$, you can use `Atn` to get a value for π .

```
Dim pi#  
End Globals  
  
pi = Atn(1) * 4  
Print pi
```

Program output:

```
3.14159265359
```



CD Example: Atn.bas

See Also:

```
Sin; Cos; Tan; Asin; Acos; Atan
```

AutoClip statement

<i>✗ Appearance</i>	<i>✓ Standard</i>	<i>✗ Console</i>
---------------------	-------------------	------------------

Syntax:

```
AutoClip [=] booleanExpr
```

Description:

If *booleanExpr* evaluates as “false” (zero), this statement turns off the “autoclip” flag for the current output window. If *booleanExpr* evaluates as “true” (nonzero), this statement turns on the “autoclip” flag for the current output window, unless the window was created with the `_noAutoClip` attribute.

If a window’s autoclip flag is “on,” things like buttons, edit fields and picture fields are excluded from the window’s “clipping region.” This exclusion means that such object are “protected”--drawing commands won’t draw over such areas, and the `CLS` command won’t erase them. While leaving autoclip “on” is normally a good idea, it can slow down the execution of drawing commands, especially on older Macs.

Any FB^3 window will have its autoclip flag “on” at the time it’s created, unless the `_noAutoClip` attribute is set in the `Window` statement that creates it.

See Also:

`_noAutoClip` attribute in the `Window` statement

Beep

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

Beep

Description:

This statement produces a system beep as defined by the Sound control panel. `Beep` is useful for alerting the user that the application needs attention.

Example:

```
For count = 1 To 5
  Beep           'exciting five-Beep mono-melody
Next
```

See Also:

Sound

Begin Enum

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

Begin Enum [start [,inc]]
    _constName1 [= staticExpression1]
    _constName2 [= staticExpression2]
    .
    .
End Enum

```

Description:

This statement begins a block of “enumerated constant” definition lines. The block must be terminated with the `End Enum` statement. All of the constants defined in this block are global, regardless of where in the program the block appears.

The `Begin Enum...End Enum` block is “non-executable,” which implies that it won’t be repeated or skipped if it appears within any kind of “conditional execution” block, such as `For...Next`, `Long If...End If`, `Do...Until`, etc. (but it can be conditionally included or excluded if it appears inside a `Compile Long If` block).

Each `_constName` represents a symbolic constant name that has not previously been defined, and each `staticExpression` represents an integer expression which consists only of:

- integer literal constants;
- previously-defined symbolic constant names;
- operators (like +, −, *, /, >, =);
- parentheses

(In particular, it can’t contain variables, nor function references.)

The `Begin Enum` block assigns values to each of the `_constName` symbolic constants as follows:

- If the `_constName` is followed by `= staticExpression`, then `_constName` is assigned the value of `staticExpression`;
- If the `_constName` is not followed by `= staticExpression`, then `_constName` is assigned the value of the `_constName` in the line above it, plus the value of `inc`;
- If the very first `_constName` is not followed by `= staticExpression`, then it’s assigned the value of `start`.

The `start` and `inc` parameters, if included, must each be a static integer expression. The default value of `start` is 0, and the default value of `inc` is 1.

Example:

In the following, the dwarves are assigned values of 1 through 7; `_snowWhite` is assigned the value 100, and `_thePrince` is assigned the value 101.

```
Begin Enum 1
    _docDwarf
    _sneezy
    _grumpy
    _sleepy
    _dopey
    _happy
    _bashful
    _snowWhite = 100
    _thePrince
End Enum
```

Begin Globals

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

Begin Globals
    [statements including variable declarations]
End Globals

```

Description:

The `Begin Globals` and `End Globals` statements indicate the beginning and end, respectively, of a section of global variable declarations. A global variable is one which is “visible” to all parts of the program: that is, it maintains its value when local function are entered or exited. By judicious placement of `Begin Globals...End Globals` statements, you can also create variables which are considered “global” to some local functions but not to others.

`Begin Globals` and `End Globals` are “non-executable” statements, so you can’t change their effect by putting them inside a conditional execution structure such as `Long If...End If`. However, you can conditionally include or exclude them from the program by putting them inside a `Compile Long If` block.

You may include any number of `Begin Globals...End Globals` pairs in your program, although typically global variables are all defined within a single section near the beginning of the program. You may also include `Begin Globals...End Globals` pairs in local functions. They must occur in matched pairs when they occur within a local function, and should normally be in matched pairs when they occur in the “main” part of your program (the “main” part consists of those lines which are outside of all local functions). When you include a `Begin Globals...End Globals` section in “main,” it should not enclose any local functions, or variables may be scoped in unpredictable ways.

When a variable’s first appearance within “main” occurs within a `Begin Globals...End Globals` section, that variable is declared as global to all local functions which appear below that section. All other variables in “main” are local to “main.” Important: FB³ places an “implicit” `Begin Globals` statement at the beginning of your program. That means that, by default, all variables declared in “main” are global. You must include an `End Globals` statement in “main” if you want any of the variables declared in “main” to be local to “main.”

When a variable’s first appearance within a local function occurs within a `Begin Globals...End Globals` section, that variable is declared as global to that function and to all local functions which appear below that function. That variable is also global in “main,” if its first appearance in “main” occurs below the function in which the variable was declared global. All other variables in the local function are local to that function, unless they were declared global in some preceding `Begin Globals...End Globals` section.

Note:

With respect to global variables, the default behavior of FB^3 is different from that of FutureBASIC II. If you include neither a `Begin Globals` nor an `End Globals` statement anywhere in your program, then the two versions behave as follows:

FutureBASIC II: All variables declared in “main” are local to “main.”

FB^3: All variables declared in “main” are global.

If you have a FutureBASIC II program which does not contain an `End Globals` statement, you should add an `End Globals` statement to the beginning of the code (before any variables are declared) in order to make it run as expected in FB^3. If your FutureBASIC II program already has an `End Globals` statement in it, you should not change it.

See Also:

`End Globals`

Begin Record

statement✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

Begin Record typeName
    recDefnBlock
End Record

```

Revision:

July 19, 2000 (Release 3)

Description:

Begins the definition of a “true” record type (as opposed to a pseudo-record type, which is defined using the `Dim Record...Dim End Record` statements). The record type definition must end with an `End Record` statement.

A `Begin Record...End Record` block is non-executable, so you can’t change its effect by putting it inside a conditional execution structure such as `Long If/End If`. However, you can conditionally include or exclude it from the program by putting it inside a `Compile Long If` block.

The record type defined in the `Begin Record...End Record` block is global in scope, and can be used anywhere below where the block appears.

typeName is a name that identifies the record type. This name must be unique among all defined record types in your program.

recDefnBlock is a block of one or more `Dim` statements. These `Dim` statements have a syntax which is identical to that of an ordinary `Dim` statement. However, instead of declaring variables, these `DIM` statements declare the names and types of the fields in this type of record. The field names do not need to be unique to this type of record (that is, a different record type could use some of the same field names as this record type). A field can be of any data type, including a previously-defined record type. A field may also be an array of elements of any type.

You can also use the following “leading dot” syntax within the record definition block, to declare “empty” space; that is, some bytes within the record which are not identified by any field name:

```
Dim .constant
```

...where `constant` is an integer literal, or a symbolic constant name (without its leading underscore character). This declares the specified number of bytes as being “nameless.”

You can also use the “semicolon” syntax after the definition of a field name, in order to specify how many bytes should be skipped between the beginning of this field and the beginning of the following field. You can use this either to insert “nameless” bytes within the record, or to make fields overlap in memory. See the `Dim` statement for more information about the semicolon syntax.

After a record type has been defined using `Begin Record...End Record`, it can be used just like any other data type. This means:

- You can declare variables as having type `typeName`;
- You can declare arrays as having type `typeName`;
- You can declare fields in other record types as having type `typeName`.

Anywhere below the `End Record` statement, you can use the `Dim` statement, along with the `As` keyword, to declare a variable, array or field of type `typeName`. For example, if you have defined a record type called `address`, then you can do the following:

```
Dim myHouse As address, yourHouse As address
Dim relatives(15) As address
Begin Record employeeInfo
    Dim 50 name$
    Dim 9 socSecNo$
    Dim 20 hobbies$(9)
    Dim empAddress As address
End Record
```

After you have declared a variable of a given record type, then you can use the “embedded dot” syntax to refer to individual fields within the record. Using the above example:

```
Dim mySecretary As employeeInfo
mySecretary.socSecNo$ = "456-78-9999"
```

Arrays of records and arrays of fields

When you use arrays of pseudo-records, you always write the array subscript at the end of the expression, whether the expression indicates an entire record or one of its fields. For example, if we have a pseudo-record array called `game`, then `game(7)` refers to element #7 in the array. If this record type has a field called `score`, then we represent the `score` of `game(7)` as:

```
game.score(7) 'pseudo-Record
```

Example:

```
Begin Record studentInfo
    Dim 20 firstName$
    Dim 20 lastName$
    Dim 1 finalGrade$
End Record
Dim myStudents(35) As studentInfo

'This represents the final grade of myStudent #14:
myStudents.finalGrade$(14) = "B"
```

Arrays inside of true records

True records have the ability to hold arrays inside of each record. This special embedded array is designated by using a bracket instead of a parenthesis for individual elements. The brackets are used for both dimensioning and accessing the sub-elements.

Example:

```
Begin Record studentInfo
  Dim 20 firstName$
  Dim 20 lastName$
  Dim grades[100]
End Record

Dim myStudents(35) As studentInfo

myStudents.grades[1](5) = 96
```

In the final line of this example, the grade element number 1 of student number 5 is set to 96.

See Also:

```
Dim Record...Dim End Record; Dim; Begin Union; SizeOf
```


Begin Union

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

Begin Record recordName
    Dim statements...
    Begin Union
        Dim statements...
    End Union
End Record

```

Description:

A union is used to set aside space in a record that may potentially contain more than one size variable. The following example sets aside two equal offsets within a record for variables of differing sizes:

```

Begin Record recordWithUnion
    Dim beForeUnion
    Begin union
        Dim inUnion1`
        Dim inUnion2$
    End union
End Record

```

```

Dim myTest As recordWithUnion
myTest.inUnion2$ = "COW"
Print myTest.inUnion1

```

The variable `myTest.inUnion1` is a single byte which occupies the same space as the first byte in the string `myTest.inUnion2$`. In this case, `myTest.inUnion1` happens to be the length byte of the string and the `Print` statement will produce "3". Such an overlap is not necessary and the two values may have no relation to one another except that they start at the same location in memory.

When FB^{^3} encounters a `Begin Union` statement, all `Dims` up to the `End Union` statement are examined and the largest item in the union determines the amount of space set aside by the compiler. In the example above, the union would occupy 256 bytes since the largest element in the union is a 256 byte Pascal string.

See Also:

```
Dim Record...Dim End Record; Dim; Begin Record
```

BeginAssem

statement✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

BeginAssem
    assembly_statements
EndAssem

```

Description:

Marks the beginning of a block of assembly-language statements. The block must be terminated with the `EndAssem` statement. The statements are assembled directly into the compiled program and executed when the program is run.

Each line in the *assembly_statements* block should be in this format:

```
[label] opCode [operands] [;remark]
```

Note that there should be at least one space character separating each of the four fields of this line. In particular, if `label` is omitted, you must include at least one leading space in front of `opCode`. But if `label` is included, then there should be no leading space to the left of `label`. To make a nicely formatted listing, use the Tab key to separate the fields.

You can also create assembly-language statements without using the `BeginAssem` and `EndAssem` statements, by preceding each assembly statement line with a “grave accent” character (that’s the character to the left of the “1” on the keyboard. It looks like a “backwards” apostrophe: ```):

```
`[label] opCode [operands] [;remark]
```

This method has the disadvantage that you can’t specify a `cpuType` for each group of lines: assembly lines that begin with the “```” mark are always assembled according to the “preferred CPU,” using the rules given above.

It’s important that you understand which CPU your assembly statements are being assembled for, and that your statements are appropriate for that CPU. You can use the `Compile Long If cpuType` statement to exclude your assembly statements from inappropriate compilations.

See Also:

`MachLg; Compile Long If`

Bin\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
binString$ = Bin$(expr)
```

Description:

This function returns a string of zeros and ones representing the binary value of *expr*, in “two’s-complement integer” format (this is the native format in which integers are stored in FB^3). If `DefStr Byte` is in effect, an 8-character string will be returned. If `DefStr Word` is in effect, a 16-character string will be returned. If `DefStr Long` is in effect, a 32-character string will be returned.

Example:

The chart below shows the results of `Bin$` on some integer values. (If a non-integer *expr* is used, `Bin$` converts it to an integer before generating the string.) The chart assumes that `DefStr Word` is in effect.

<i>expr</i>	Bin\$(<i>expr</i>)
1	0000000000000001
-1	1111111111111111
256	0000000100000000
-256	1111111100000000

CD Example: `Bin$.bas`**Note:**

To convert a string of binary digits into an integer, use the following technique:

```
intVar = Val("&X" + binaryString$)
```

`intVar` can be a (signed or unsigned) byte variable, short-integer variable or long-integer variable. Byte variables can handle a `binaryString$` up to 8 characters in length; short-integer variables can handle a `binaryString$` up to 16 characters in length; long-integer variable can handle a `binaryString$` up to 32 characters in length.

See Also:

`Hex$`; `Oct$`; `Uns$`; `DefStr Byte/Word/Long`; *Appendix C: Data Types and Data Representation*

Bit**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
bitValue = Bit(bitPos)
```

Description:

This function returns an integer whose binary representation has the bit in position *bitPos* set to “1”, and all other bits set to “0”. Bit positions are counted from right to left: a *bitPos* value of zero corresponds to the rightmost (“least significant”) bit. The maximum allowable value for *bitPos* is 31, which corresponds to the leftmost bit in a long-integer value. **Bit** is useful in conjunction with “bitwise operators” like **And** and **Or**, for setting and testing the values of particular bits in a quantity.

Note:

If *_bitPos* is a symbolic constant name, then you can use *_bitPos%* (note the “%”) as a synonym for **Bit**(*_bitPos*).

Example:

CD Example: Bit.bas

The following expression evaluates as *_zTrue* (−1) if bit “n” is set in *testValue&*:

```
(testValue& And Bit(n)) <> 0
```

The following assignment sets bit “n” in *testValue&* to 1:

```
testValue& = (testValue& Or Bit(n))
```

The following assignment resets bit “n” in *testValue&* to 0:

```
testValue& = (testValue& And Not Bit(n))
```

See Also:

Bin\$; And; Or; Not; Appendix C: Data Types and Data Representation

BlockMove

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
BlockMove sourceAddr &, destinationAddr &, numberBytes &
```

Description:

Copies a range of bytes which begin at address *sourceAddr* &, to the address range that begins at address *destinationAddr* &. The *numberBytes* & parameter specifies the number of bytes which are to be copied. The copying works correctly even if the source and destination ranges overlap.

Example:

```
Dim src%(79999), dst%(79999)           `80000-element arrays
`Copy one array To the other:
BlockMove @src%(0), @dst%(0), 160000
```



CD Example: BlockMove.bas

Note:

For copying small amounts of data (80 bytes or less) into a variable, it is more efficient to use the `var;length = address` syntax of the `Let` statement. See the `Let` statement for more details.

See Also:

`Let`; `VarPtr`

Box**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Box [Fill] [h, v] To h1, v1 [To h2, v2 ...]
```

Description:

Draws a rectangle which has diagonally opposite corners at coordinates (*h*, *v*) and (*h1*, *v1*). The rectangle's frame is drawn using the current pen size, mode, pattern and color for the current window or printer. If the `Fill` parameter is specified, then the rectangle is filled with the current window pattern.

If the `To` keyword is specified more than once, then several rectangles are drawn, each using (*h*, *v*) as one corner, and the coordinates following `To` as the diagonally opposite corner.

If the *h* and *v* parameters are omitted, then the rectangle's originating point is set to one of the following:

- The (*h*,*v*) coordinates of the most recent `Box` statement (in any window) that actually specified the *h* and *v* parameters;
- The last point specified in the most recent `Plot` statement (in any window);
- (0,0), if no `Plot` statement has yet been executed, and no prior `Box` statement that specified *h* and *v* has yet been executed.

Example:

CD Example: `Box.bas`, `Box#2 Bar Chart.bas`

Console behavior:

When you use the Console runtime, `Box` always switches to the Graphics Window before drawing. You can't use `Box` to draw a box in the Text Window, or on the printer; use the Toolbox procedures `FrameRect` or `PaintRect` instead. Alternatively, you can activate the graphics window and select Print from the File menu.

See Also:

`Circle`; `Plot`; `Pen`

Button**function**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
butTonState = Button[(ID [,selector])]
```

Revision:

February, 2002 (Release 6)

Description:

Returns the state of the button specified by *ID* (in the current window), or the scroll value of the scroll bar specified by *ID* (in the current window). If *ID* is omitted, the function returns the previous value of the scroll bar that was most recently clicked.

For standard buttons, `Button(ID)` returns one of the following values:

<i>Constant</i>	<i>Value</i>	<i>Meaning</i>
<code>_grayBtn</code>	0	Button is dimmed and inactive
<code>_activeBtn</code>	1	Button is active (clickable). If a checkbox, it's not checked. If a radio button, it's not filled.
<code>_markedBtn</code>	2	Button is active (clickable). If a checkbox, it's checked. If a radio button, it's filled.

The extended button function offers access to many of the features found in Appearance Manager buttons. Use the selectors in conjunction with the button reference number to obtain information. For example, to get the minimum value for button number 10, the code would be:

```
min = Button(10, _FBGetCtlMinimum)
```

If the `Button` statement is called and the *ID* parameter refers to a button that does not exist, you will see the message, "Button() called for non-existent button." If an improper selector is used, you will see the message, "Bad parameter for Button()." The following table lists possible values for the button statement's selector.

<i>selector</i>	<i>Description</i>
< zero	Get the reference number of the nth embedded sub control. The absolute value of this is used as the index. Example: subControlRef=Button(_mySuperControl,-3) Rem get 3rd embedded control's ref num
_FBGetCtlRawValue	Get current value for control. Unlike FB's standard Button function which adds 1 to the control, this version passes the raw value to the caller without alteration.
_FBGetCtlMinimum	Get the minimum value allowed for this control.
_FBGetCtlMaximum	Get the maximum value allowed for this control.
_FBGetCtlPage	Get page up/down value for a scroll bar.
_FBGetRootControl	Get the root control of the window.
_FBCountSubControls	Count how many controls are embedded in this super control.
_FBGetSuperControl	Get (parent) super control's ref num.
_FBGetControlDate	Fills the global Pascal string gFBControlText\$ with the text version of the control's date. It also fills the global record gFBControlSeconds with the control's date/time record. See "Time and Date Buttons" under the Appearance Button statement.
_FBGetControlTime	Fills the global Pascal string gFBControlText\$ with the text version of the control's time. It also fills the global record gFBControlSeconds with the control's date/dime record. See "Time and Date Buttons" under the Appearance Button statement.
_FBGetControlTEHandle	Get the handle to the text field of a button that contains a text field.
_FBGetBevelControlMenuHandle	Gets the menu handle of a beveled pop-up menu button.
_FBGetBevelControlMenuVal	Gets the current selection of a beveled pop-up menu button.
_FBGetBevelControlLastMenu	Gets the previous selection of a beveled pop-up menu button.
_FBGetControlMenuHandle	Gets the menu handle of a standard pop-up menu button.
_FBGetControlMenuID	Gets the current selection of a standard pop-up menu button.

See Also:

```
Def BtnRect; Usr Handle2Btn; Button; Scroll Button
```


Button&**function**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
btnHandle& = Button&(btnID&|%)
```

Revision:

February, 2002 (Release 6)

Description:

For the button or scroll button specified by *btnID%* or *btnID&* (for the appearance runtime) in the current window, this function returns a handle to the button's "control record." The control record contains information about a button's rectangle, state, title, etc. (see the Control Manager chapter in Inside Macintosh: *Macintosh Toolbox Essentials*, for complete details). Given the handle, you can access the various fields of the control record as follows:

<i>btnHndl&..nextControl&</i>	handle to next control in window
<i>btnHndl&..ctrlOwner&</i>	pointer to control's window
<i>myRect;8 = [btnHndl&]+_ctrlRect</i>	control's rectangle
<i>btnHndl&..ctrlVis`</i>	255 if control visible
<i>btnHndl&..ctrlHilite`</i>	current highlight state
<i>btnHndl&..ctrlValue%</i>	control's current setting
<i>btnHndl&..ctrlMin%</i>	control's minimum setting (scrollbars)
<i>btnHndl&..ctrlMax%</i>	control's maximum setting (scrollbars)
<i>btnHndl&..ctrlDefHandle&</i>	handle to control definition function
<i>btnHndl&..ctrlData&</i>	data accessed by ctrl definition function
<i>btnHndl&..ctrlAction&</i>	address of default action procedure
<i>btnHndl&..ctrlRfCon&</i>	control's RefCon field (see below)
<i>btnHndl&..ctrlTitle\$</i>	control's title

FB^3 (standard runtime) uses the RefCon field (*btnHndl&..ctrlRfCon&*) to maintain information about the control, including control type, ID number, "enable" flag, and for scroll bars, the page increment value. This information is only valid for controls created by the `Button` or `Scroll Button` statements. You can access this information as follows:

```
refCon& = btnHndl&..ctrlRfCon&
controlType = (refCon& And &E0000000) >> 29      `bits 29-31
controlID   = (refCon& And &1FFF0000) >> 16      `bits 16-28
disabled    = (refCon& And &8000) <> 0           `bit 15
pageInc     = (refCon& And &7FFF)                `bits 0-14
```

Note:

In Carbon, the technique shown above no longer works because the `ControlRecord` structure has become opaque. The fields of such a structure are accessible through specific Toolbox functions, usually called accessors (getter and setter). Those functions are also available for PPC compile.

`nextControl` see example “Print window's content” on the CD.

```

ctrlOwner = Fn GetControlOwner( btnHndl& )

rectPtr& = Fn GetControlBounds( btnHndl&, ctrlRect )
Call SetControlBounds( btnHndl&, ctrlRect )

```

Associated functions :

```

Call MoveControl( btnHndl&, horizontal, vertical )
Call SizeControl( btnHndl&, largeur, hauteur )

ctrlVis = Fn IsControlVisible( btnHndl& )
err = Fn SetControlVisibility( btnHndl&, ctrlVis, doDraw )

```

Associated functions:

```

Call ShowControl( btnHndl& )
Call HideControl( btnHndl& )

ctrlHilite = Fn GetControlHilite( btnHndl& )
isCtrlHilite = Fn IsControlHilited( btnHndl& )
Call HiliteControl( ControlRef, hiliteState )

err = Fn ActivateControl( btnHndl& )
err = Fn DeactivateControl( btnHndl& )
isCtrlActive = Fn IsControlActive( btnHndl& )

ctrlValue = Fn GetControlValue( btnHndl& )
Call SetControlValue( btnHndl&, ctrlValue )

ctrlMin = Fn GetControlMinimum( btnHndl& )
Call SetControlMinimum( btnHndl&, ctrlMin )

ctrlMax = Fn GetControlMaximum( btnHndl& )
Call SetControlMaximum( btnHndl&, ctrlMax )

ctrlData& = Fn GetControlDataHandle( btnHndl& )
Call SetControlDataHandle( btnHndl&, ctrlData& )

ctrlAction& = Fn GetControlAction( btnHndl& )
Call SetControlAction( btnHndl&, ctrlAction& )

ctrlRfCon& = Fn GetControlReference( btnHndl& )
Call SetControlReference( btnHndl&, ctrlRfCon& )

Call GetControlTitle( btnHndl&, ctrlTitle$ )
Call SetControlTitle( btnHndl&, ctrlTitle$ )

```

FB^3 Appearance Manager runtime uses the RefCon field to store a handle to additional information. The format of that handle may be displayed in the following record format (from the RnTm Appearance.GLBL header file):

```

Begin Record FBcontrolDescription
  Dim FBcontrolRef      As Long // standard FB button,
                               // or EF, or picField,
                               // ref number

  Begin union
    Dim FBcontrolMin    As Long
    Dim FBefJustClass   As Long
  End union

  Dim FBcontrolMax      As Long
  Dim FBcontrolInitVal  As Long
  Dim FBTempRefCon      As Long // CDEF requires special
                               // refcon

  Dim FBcontrolPage     As Long // for Scroll bars
  Dim FBefLinkcHndl     As Handle // Scroll bar cHndl for ef
                               // (0 if none)

  Begin union
    // TEHandle of pane or of linked pane (0 if none)
    Dim FBteHndl        As Handle
    Dim FBpictHndl      As Handle // PF picture Handle
  End union
  Dim FBcustOnEdit      As Ptr // for EF custom On Edit
  Dim FBcontrolClass    As Short // button, EF, picture etc
  Dim FBcntrlSubclass   As Short // kind of button or
                               // kind of EF

  Dim FBefBackPat       As RGBColor
  Dim FBefBackColor     As RGBColor
  Begin union
    Dim FBpfJust        As Byte
    Dim FBefReturnOK    As Boolean
  End union
  Begin union
    Dim FBtransparent   As Boolean // for PF
    Dim FBhasFocus      As Boolean // for EF pane Handlers
  End union
  Begin union
    Dim FBteActive      As Boolean // see EFActivateText
    Dim FBclickReverse  As Boolean // for PF
  End union
  Dim FBNoDrawFocus     As Boolean // see FBdrawFocusAndFrame
  Dim FBcntrlFramed     As Byte // for EF&PF pane Handlers
  Dim FBpaletteEF       As Boolean // field is on a palette
  // grayishTextOr when window inactive or EF disabled ==
  // pane inactive
  Dim FBefAuToGray      As Boolean
  Begin union
    Dim FBefDisabled    As Boolean // keeps disabled
                               // EF grayishTextOr
    Dim legacyValue     As Boolean // old Button syntax
  End union
  Dim &
End Record

```

Note:

Your program should not dispose of the handle returned by the `Button&` function (i.e., don't execute `Def DisposeH(btnHndl&)`). The handle is automatically disposed when you close the button (`Button Close`) or close its window.

See Also:

`Def BtnRect; Usr Handle2Btn; Button; Scroll Button`

Button

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

To create a button:

```
Button [#] btnID, state, title, btnRect [, btnType]
```

To alter a button:

```
Button [#] btnID [, [state] [, [title] [, btnRect]]]
```

To hide a button (Appearance Manager only):

```
Button [#] -btnID
```

Revision:

February, 2002 (Release 6)

Description:

The `Button` statement puts a new button in the current output window, or alters an existing button's characteristics. After you create a button using the `Button` statement, you can use the `Dialog` function to determine whether the user has clicked it. You can use the `Button Close` statement if you want to dispose of the button without closing the window.

With the Appearance Manager, you may hide an existing button using the `Button` statement with a negative button reference number. You can deactivate the control with either:

```
Button 1, _grayBtn
```

or ...

```
Appearance Button 1, _grayBtn
```








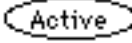




When you first create a button with a given ID (in a given window), you must specify all the parameters up to and including *btnRect* (the *btnType* parameter is optional; it defaults to `_push`). If you later want to modify that button's characteristics, execute `Button` again with the same ID, and specify one or more of the other parameters (except *btnType*, which cannot be altered). The button will be redrawn using the new characteristics that you specified; any parameter that you don't specify will not be altered.

<i>Parameter</i>	<i>Description</i>								
<i>btnID</i>	An ID number in the range of 0...8191 that you assign when you create the button and that you refer to when altering or closing the button. The number you assign must be different from the ID's of all other existing buttons and scroll bars in the current window.								
<i>state</i>	Sets the state of the button. See "Button function" for details								
<i>title\$</i>	The text that appears inside the button (in the case of push buttons) or to the right of the button (in the case of checkboxes and radio buttons).								
<i>btnRect</i>	The button's enclosing rectangle. This can be specified in either of two ways: <div style="display: flex; align-items: center;"> <div style="margin-right: 20px;"> $(x1, y1) - (x2, y2)$ @rectAddr& </div> <div> Coordinates of two diagonally opposite points Address of an 8 byte rectangle structure. </div> </div>								
<i>btnType</i>	<p>Specifies the type of button:</p> <table style="border: none;"> <tr> <td><code>_push</code></td> <td>(1) push button (default type)</td> </tr> <tr> <td><code>_checkBox</code></td> <td>(2) check box</td> </tr> <tr> <td><code>_radio</code></td> <td>(3) radio button</td> </tr> <tr> <td><code>_shadow</code></td> <td>(4) framed push button</td> </tr> </table> <p>If you add the constant <code>_useWFont</code> to any of the above types (except <code>_shadow</code>) the button's title will be drawn using the window's current font ID, font size, and font style. Any subsequent change you make to the window fontID, font size, or font style will be reflected in the button's title when it is redrawn. If you don't specify <code>_useWFont</code> (or if the button type is <code>_shadow</code>) the title is drawn using the system font.</p>	<code>_push</code>	(1) push button (default type)	<code>_checkBox</code>	(2) check box	<code>_radio</code>	(3) radio button	<code>_shadow</code>	(4) framed push button
<code>_push</code>	(1) push button (default type)								
<code>_checkBox</code>	(2) check box								
<code>_radio</code>	(3) radio button								
<code>_shadow</code>	(4) framed push button								

Using FutureBASIC's special Control Definition (CDEF)

You can customize the appearance and action of buttons by including special code resources called CDEF's within your application's resource fork. FB^3 provides one such CDEF which draws several variants of buttons. To include this CDEF in your application's resource fork, specify the `_IncludeCDEF` flag in the `Compile` statement.

You can create buttons that use FutureBASIC's custom CDEF by specifying special values in the *btnType* parameter, as summarized in this table:

<i>btnType</i>	<i>Type of button drawn :</i>		
_CDEFBaseID + _push _CDEFBaseID + _checkbox _CDEFBaseID + _radio _CDEFBaseID + _shadow	These btnType values produce buttons which are similar to the system button types but which use Geneva 9 point text rather than the system font.		
_CDEFBaseID+_push+_CDEFnoOutline			
_CDEFBaseID+_checkbox+_CDEFnoOutline			
_CDEFBaseID+_radio+_CDEFnoOutline			
_CDEFBaseID+_shadow+_CDEFnoOutline			

FutureBASIC's built-in custom CDEF uses the 9-point Geneva font by default. If you add the constant `_useWFont` to any of the button types in the table above, the button will use the font ID, font size and style which are current at the time the button is created. Unlike the standard "system" buttons, `_CDEFBaseID` buttons which specify `_useWFont` will *not* change their font characteristics after they've been created, even if the window's font changes.

Using your own custom CDEF's

If you have a custom CDEF resource of your own that you wish to include in your application, copy the CDEF into a resource file which is referenced in your program's `Resources` statement. Your CDEF's resource ID should not be zero (which is reserved for the System CDEF) nor 8 (which is the resource ID that FutureBASIC's special CDEF uses). To create a button that uses your custom CDEF, use a *btnType* value that is calculated according to the following formula:

$$\text{btnType} = \text{CDEFresourceID} * 16 + \text{butTonVariant} + 1$$

When your custom button is created, the values you specify in the *title\$* and *btnRect* parameters will be stored into the `ctrlTitle` and `ctrlRect` fields of the button's control record. If you specify a *state* value of zero, then the control record's `ctrlValue` field will be set to zero and the `ctrlHilite` field will be set to 255. If you specify any other value for *state*, then the `ctrlValue` field will be set to *state*-1, and the `ctrlHilite` field will be set to zero. See the Control Manager chapter of Inside Macintosh: *Macintosh Toolbox Essentials* for more information.

See Also:

AutoClip; Button& function; Button function; Button Close; Scroll Button;
Dialog function

Button Close

statement✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```

Button Close [#]btnID
Button Close [#]-btnID

```

Revision:

May 5, 2000 (Release 3)

Description:

The button or scroll bar specified by *btnID* is removed from the current output window. This is one of two ways you can dispose of a button: the other way is to close the window, which automatically closes all the buttons and scroll bars in it.

If `Button Close` uses a negative *btnID*, the area previously occupied by the control is not erased or invalidated. No update event will occur as a result of close a button with a negative *btnID*.

Example:

```

Button Close #1
Button Close #_radioBtn2
Button Close -1

```

See Also:

Button; Scroll Button; Appearance Button

ButtonTextString\$**function**✓ *Appearance*✗ *Standard*✗ *Console***Syntax:**

```
pString$ = Fn ButtonTextString$ (bRefNum&)
```

Revision:

February, 2002 (Release 6)

Description:

This Appearance Manager function extracts the text from a button. This is most useful for extracting information from Appearance Manager, control-based edit fields.

```
Appearance Button 1,1,1,0,1,"",¬
(10,10)-(200,200),_kControlEditTextProc

// put text into Edit Field Button
Def SetButtonTextString( 1, "Hello" )

// extract text from Edit Field Button
pString$ = Fn ButtonTextString$(1)

// pString$ now equals "Hello"
```

See Also:

```
Def SetButtonData, Button function, Appearance Button,
Def SetButtonTextString
```


Call <toolbox>**statement**✓X *Appearance*✓X *Standard*✓X *Console***Syntax:**

```
[Call] ToolboxProcName [modifiers] ¬
    [ ( [ { #addrExpr1&|arg1} [, { #addrExpr2&|arg2} ... ] ] ) ]
```

Description:

This statement calls a Toolbox procedure as defined in Inside Macintosh. A Toolbox procedure (as opposed to a Toolbox function) performs an operation without returning a value. To execute a Toolbox function, use the `Fn` statement instead.

ToolBoxProcName must be the name of a Macintosh Toolbox procedure. FB³ recognizes the names of hundreds of ToolBox procedures and functions; advanced programmers can also use the `Toolbox` statement and the `TBAlias` statement to add new Toolbox function/procedure names.

modifiers is a list of one or more constants which may modify the procedure's operation. Modifiers are not applicable to every Toolbox procedure: see the procedure's description in Inside Macintosh for more details. The modifiers include:

<code>_async</code>	execute the routine asynchronously
<code>_clear</code>	initialize a handle or pointer to nulls
<code>_sys</code>	use the System heap

If the procedure requires parameters, include them in a list surrounded by parentheses. (If the procedure does not take any parameters, then the parentheses are optional.) Depending on the specific procedure, FB³ expects to see the following kinds of things in the parameter list:

1. A short integer value;
2. A long integer value;
3. A pointer to a memory location (often, a pointer to an FB³ variable);
4. A literal string, or a variable representing a string;
5. A variable representing a record;
6. A variable which is to receive a value from the procedure.

In cases 4, 5 and 6, the Toolbox expects to receive an address that points to the string literal or to the variable; FB³ determines the address (the location in memory) of the variable or the string that appears in the list, and passes that address to the Toolbox. But there may be cases in which you may wish to explicitly specify some address (for example, you may want the Toolbox to receive an address to a buffer that doesn't correspond to any FB³ variable). In such cases, you should use the `#addrExpr&` syntax. The “#” symbol instructs FB³ to evaluate the expression in *addrExpr&* as a long integer, and to pass that long integer value directly to the Toolbox.

Example:

```
Dim rect.8  
Call SetRect(rect, 10,10, 200,150)
```

In this case, FB^3 reserves memory locations 18234650 through 18234657 (for example) for the “rect” structure. When Call SETRect is executed, FB^3 passes 18234650 to the Toolbox as the first parameter.

```
Call SetRect(#[pictH&]+_picFrame, 10,10, 200,150)
```

In this case, we want to affect the rectangle which is located at address [pictH&]+_picFrame. Because we have used the “#” symbol, FB^3 evaluates the expression [pictH&]+_picFrame. If it evaluates to, say, 26219554, then FB^3 passes 26219554 to the Toolbox as the first parameter.

See Also:

```
Fn <Toolbox>
```

Case**statement**

See the `Select Case` statement.

Chr\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
character$ = Chr$(expr)
```

Description:

Returns a 1-character string consisting of the character whose ASCII code is given by *expr* Mod 256. ASCII (which stands for American Standard Code for Information Interchange) is a system of codes for characters used on microcomputers.

Example:

```
Print Chr$(65); Chr$(66); Chr$(67)
```

Program ouptut:

ABC



CD Example: Chr\$ Demo.bas

See Also:

Asc; String\$; Space\$; Appendix F: *ASCII Character Codes*

Circle

statement

✓*Appearance*✓*Standard*✓*Console*

Syntax:

```
Circle [Fill] x, y, radius [{To | Plot} startAngle, angleSize]
```

Description:

Draws a circle, an arc or a wedge in the current foreground color, pen pattern and pen size. If a circle or wedge is drawn using the `Fill` keyword, the circle or wedge will be filled with the current pen pattern. When used in combination with the `Ratio` statement, `Circle` can be used to draw ellipses and elliptical arcs and wedges.

If the current `Ratio` is set to 0,0 (the default), then `Circle` behaves as follows:

If only the *x*, *y*, and *radius* parameters are specified, then a complete circle is drawn, with its center at point (*x*, *y*) and having a radius of *radius* pixels.

If the `To` keyword is specified, then a wedge (two radii plus an arc) is drawn. The first radius is drawn in the orientation specified by *startAngle*, which is measured in units of “brads” (see below). Angles are measured counterclockwise starting from the “3-o’clock” position, which corresponds to zero brads. The *angleSize* parameter specifies the angular width of the wedge (also in brads); the wedge always extends counterclockwise from the *startAngle* position. Note that the width of the “wedge” may be greater than a half-circle, in which case the “wedge” looks more like a Pac-Man.

If the `Plot` keyword is specified, then an arc is drawn without any radii. The position and size of the arc are the same as when the `To` keyword is specified. If both the `Plot` keyword and the `Fill` keyword are specified, then the `Circle` command does nothing.

“Brads” are an angular unit in which a full circle corresponds to 256 brads. A brad is therefore slightly larger than a degree (to be exact, it’s 360/256 of a degree). A half circle therefore equals 128 brads, and a right angle equals 64 brads.

Example:



CD Example: Circle.bas

Console Behavior:

When you use the Console runtime, `Circle` always switches to the Graphics Window before drawing. You can’t use `Circle` to draw a circles and arcs in the Text Window, or on the printer; use the Toolbox procedures `FrameOval`, `FrameArc` or `PaintOval` instead. Alternatively, you can activate the graphics window and select Print from the File menu.

Note:

If you use values outside the range 0..255 for *startAngle* and/or *angleSize*, then values modulo 256 will be used.

See Also:

Ratio; Fill; FrameOval and FrameArc Toolbox procedures

Clear

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

`Clear`

Description:

Sets all global and “main” program variables (including array elements) to zeros or null strings. Variables defined in local functions are not affected (see the `Local` statement).

See Also:

`[Clear] Local` statement

Clear <index>**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

To allocate, increase or decrease memory for Index\$ arrays:

1. **Clear** *bytes& [,indexID]*
2. **Clear** *numElements&, indexID, elementSize*

To release the memory used by existing Index\$ arrays:

3. **Clear** -1
4. **Clear Index\$** [*indexID*]

Description:

Use syntax (1) or (2) to allocate memory for an Index\$ string array, and to specify whether the array shall consist of fixed-length or variable-length cells for the strings. Using fixed-length cells provides greater speed but may require more memory. Using variable-length cells provides more efficient use of memory, but processing may be slower. You must use Syntax (1) or (2) to allocate memory for an Index\$ array before the array will function.

Use Syntax (1) to specify the total number of bytes to be allocated for a variable-cell-length Index\$ array. The *indexID* parameter indicates which of the ten available Index\$ arrays (numbered 0 through 9) to allocate space for. If you omit this parameter, space will be allocated for Index\$ array #0.

Use Syntax (2) to specify the number of elements to be allocated for a fixed-cell-length Index\$ array. The *elementSize* parameter should be in the range (1..256); it fixes the size of the cells. You can store a string of up to (*elementSize* - 1) characters in such a cell. The *indexID* parameter indicates which Index\$ array (0 - 9) to allocate space for.

You can also use Syntax (1) or (2) to increase or decrease the amount of memory allocated for an existing array. This is sometimes useful if you find you need more (or less) memory than you originally allocated. If you increase the memory allocation, none of the existing strings in the array will be affected. If you want to decrease the memory allocation without affecting the existing strings, make sure you don't specify too small a number; you can use the Mem function to determine how much memory the existing strings occupy.

Use Syntax (3) or (4) to release the memory previously allocated for one or more Index\$ arrays. Syntax (3) releases the memory occupied by all existing Index\$ arrays. Syntax (4) empties each string in an INDEX\$ array without releasing memory occupied by the array. If you omit the *indexID* parameter, then Index\$ array #0 is used.

See Also:

Index\$; Index\$ D; Index\$ I; IndexF

Clear Local

See the `Local` statement.

statement

Clear LPrint

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------


Syntax:

```
Clear LPrint
```

Description:

If output has been routed to the printer, `Clear LPrint` forces the Printing Manager to print the current page, without closing the print job. The other ways to print the current page are to execute `Close LPrint` or to exit the program, both of which also cause the print job to be closed.

Example:

 CD Example: Printer.bas

See Also:

```
Page; Close LPrint; Def LPrint; Def Page; Route _toPrinter;  
Route _toScreen
```

Close

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Close [[#]deviceID1 [, [#]deviceID2 ...]]
```

Description:

Closes one or more devices (usually a serial port or a disk file) previously opened with the `Open` statement. If no `deviceID` is specified, all open devices are closed.

Closing a file releases a certain amount of memory, may allow other processes to access the file, forces any remaining bytes in the output buffer to be written to disk, and allows you to re-use the number specified in `deviceID` (for a subsequent `Open` statement)

See Also:

```
Open; Reset
```

Close Folder

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Close Folder wdRefNum%
```

Description:

This statement causes the specified working directory reference number to be removed from the System’s list of current working directories.

A working directory reference number is a temporary number which is used to uniquely identify a certain directory (folder) on a certain currently-mounted volume. These numbers are generated by the System software on request; several FB^3 functions (such as the Files\$ function) generate and return working directory reference numbers. A folder is considered to be “open” once a working directory reference number has been generated for it.

The System’s list of current working directories has a limited amount of space available; executing Close Folder may in some cases help prevent the list from becoming full. In addition, a folder cannot be deleted from disk while it remains on the list: you may therefore find it necessary to execute Close Folder if you intend to delete a folder.

After Close Folder has been executed, the working directory reference number is no longer valid, and can no longer be used to identify that folder.

See Also:

```
Files$
```

Close LPrint

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Close LPrint
```

Description:

After output has been routed to the printer, `Close LPrint` informs the Print Manager that the print job is complete. The current page is printed and the print job closed. You should execute a `Route _toScreen` statement immediately before or after calling `Close LPrint`.

Example:



CD Example: Printer.bas

See Also:

```
Clear LPrint; Def LPrint; Def Page; Route _toPrinter, Route _toScreen
```

Cls statement

✓*Appearance*✓*Standard*✓*Console***Syntax:****Cls** [**Line** | **Page**]**Description:**

Cls resets the current window's clipping region to encompass the entire window, clears the entire contents of the window to the background pattern and color (usually white), and resets the pen to a position near the upper-left corner (0, 0) of the window. If the window contains controls or edit fields created by FB^3 statements, and the window's "autoClip" attribute is turned on, then Cls will exclude those areas from the clipping region, and will not erase them.

Cls *Line* clears a rectangle as high as the current font, from the current pen position to the right side of the window. This is handy for clearing text from the current line only. The pen position is not affected.

Cls *Page* clears the text from the pen position to the right side of the window (as Cls *Line* does), then clears the entire window below the pen position. The pen position is not affected.

Console behavior:

When you use the Console runtime, Cls switches to the Text Window before executing; therefore it only clears text. If you want to clear the Graphics Window, first switch to the Graphics Window (using a statement like Pen), then call the Toolbox procedure `EraseRect`.

See Also:

"_auToClip" attribute of the Window statement

Color statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Color [=] colorExpr
```

Description:

The `Color` statement sets the current window's foreground color to one of eight old-style "Basic QuickDraw" colors. *colorExpr* should equal one of the following:

```
0  (_zWhite)
1  (_zYellow)
2  (_zGreen)
3  (_zCyan)
4  (_zBlue)
5  (_zMagenta)
6  (_zRed)
7  (_zBlack)
```

Console behavior:

When you use the Console runtime, `Color` switches to the Graphics Window before executing; you can't use it to change the foreground color in the Text Window nor on the printer. To change the color of printed output while running the Console, use the Toolbox procedure `RGBForeColor`.

Note:

The `Color` statement does not change the appearance of anything that's already in the window; the new color will appear the next time you draw text or a QuickDraw shape in the window.

For greater control over the foreground color, use the `Long Color` statement.

See Also:

`Long Color`

Color Window

statement

*✗ Appearance**✗ Standard**✗ Console*

Syntax:

```
Color Window {_true | _false}
```

Description:

This statement does nothing in FB^3, but is retained for backwards compatibility with programs written in FutureBASIC II. In FB^3, all windows are opened as color windows.

Compile

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Compile [tickCount][, compileFlags]
```

Description:

While it is possible to place your own compile statements in source code, this is better accomplished by selecting items in the editor's preference window. Placing `Compile` statements in source code will void most of the selections made in the editor.

The `Compile` statement is used to change the way a program is compiled. The statement is optional: if you use it, it should not appear more than once in your program's source code, except possibly within different branches of a `Compile Long If` block. It is a non-executable statement, so you can't change its effect by putting it inside a conditional execution structure such as `Long If...End If`. However, you can conditionally include or exclude it from the program by putting it inside a `Compile Long If` block.

Typically, the `Compile` statement is the first statement in the program.

tickCount sets the preferred number of ticks to occur between operating system calls.

compileFlags is a list of one or more symbolic constant names (separated by spaces) chosen from the set described below. *compileFlags* can also consist of a single integer constant equal to the sum of the individual symbolic constants.

Flag	Value	Description	Default
<code>_pointerVars</code>	1	Allocate a pointer in the application heap for variables (only when compiling a code resource).	Allocate a handle in the application heap.
<code>_sysHeapVars</code>	2	Allocate a handle in the System heap for variables (for code resource only).	Allocate a handle in the application heap.
<code>_macsBugLabels</code>	4	Generate information necessary for MacsBug to display your program's labels and function names. This is a useful feature for debugging, but makes your compiled code slightly larger.	No MacsBug information is generated.
<code>_strResource</code>	8	Store all literal strings found in the program as STR# resources.	No STR# resources are created.
<code>_dimmedVarsOnly</code>	32	Generate an error if any variable is encountered before it is explicitly declared with a Dim statement.	No Dim req'd for non-array, non-record vars (implicitly declared by appearance in code).
<code>_noRedimVars</code>	64	Generate an error if a Dim statement contains a variable that was previously encountered in the same scope (possibly in a previous Dim statement).	Any Dim with a previously encountered variable is ignored.
<code>_dontOptimizeStr</code>	256	Prevents the compiler from removing duplicate strings when creating STR# resources in an application. This only applies if the <code>_strResource</code> flag is also set.	Compiler removes duplicate strings.
<code>_chgConfig</code>	32768	Instructs the compiler to override the options specified in "Preferences" dialog when compiling this program (see the next table).	The current "Preferences" options are used.
<code>_includeCDEF</code>	65536	Add custom CDEF resource (for Geneva 9-point text or window text) to compiled applications.	Don't add the resource.
<code>_includeMDEF</code>	262144	Add custom MDEF resource (for Geneva 9-point menu text) to compiled application.	Don't add the resource.
<code>_includeWDEF</code>	524288	Add custom WDEF resource (for windoid- style palettes) to compiled application.	Don't add the resource.
<code>_ToobloxCallRequired</code>	268435456	Generates an error if references to Toolbox procedures are not preceded by the Call keyword.	Call keyword is optional.
<code>_ToolboxCallNotRequired</code>	536870912	In Toolbox procedure references, the Call keyword is optional. Same as default.	Call keyword is optional.

Example:

The following example tells the compiler to generate program labels and function names for MacsBug, and to require that all variables be explicitly declared. Tickcount is ignored if the code is used in an application.

```
Compile 0, _MacsBugLabels _dimmedVarsOnly
```

See Also:

```
Resources; Compile Long If
```

Compile End If

statement

See the `Compile Long If` statement

Compile Long If

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

Compile Long If condition
    [statementBlock1]
[Compile Xelse
    [statementBlock2]]
Compile End If

```

Description:

You can use the `Compile Long If` statement to conditionally include or exclude selected lines of code from the compiled version of your program. This is useful if you need to maintain two (or more) slightly different versions of your program; `Compile Long If` allows you to maintain them both within the same source file.

If the condition following `Compile Long If` is evaluated as “true” or non-zero, then the statements in *statementBlock1* are included in the compilation and the statements (if any) in *statementBlock2* are ignored by the compiler. If the condition is evaluated as “false” or zero, then the statements in *statementBlock1* are ignored by the compiler, and the statements (if any) in *statementBlock2* are included in the compilation.

condition must be in one of the following forms:

- *constExpr*
- {Def | NDEF} *_symbolicConstant*
- Tron [= *_false*]
- cpu68K
- cpuPPC
- carbonLib
- A list of any of the above separated by “And” or “Or”. You may also optionally surround any valid sub-condition with parentheses.

constExpr is a “static integer expression.” A static integer expression is any valid expression which consists of only:

- integer literal constants;
- previously-defined symbolic constant names;
- operators (like +, -, *, /, >, =);
- parentheses

(In particular, it can’t contain variables, nor function references.) If you use this form of `Compile Long If`, then the condition will be evaluated as “true” if the expression’s value is nonzero.

`_symbolicConstant` stands for a symbolic constant name. `Def _symbolicConstant` is evaluated as “true” if the indicated constant was previously defined. `NDEF _symbolicConstant` is evaluated as “true” if the indicated constant was not previously defined.

The condition `Tron` is evaluated as “true” if debugging has been activated for the current section of code. The condition `Tron = _false` is evaluated as “true” if debugging has not been activated for the current section of code. (See the `Tron` statement for more details.)

If you use the keyword `cpu68K` as the condition, it’s evaluated as “true” if the current compilation is being compiled into Motorola 680x0 (“68k”) machine code. If you use the keyword `cpuPPC` as the condition, it’s evaluated as “true” if the current compilation is being compiled into PowerPC machine code.

Example:

Because `Compile Long If` can cause lines (including non-executable lines) to be completely ignored by the compiler, you can use it to control such things as the declaration of variables, program labels, constants, and even entire functions. For example:

```

Compile Long If _needBigArray
  Dim myArray$(3000)
Compile Xelse
  Dim myArray$(30)
Compile End If

Compile Long If _dimensions = 3
  Def Fn Diagonal!(a!, b!, c!) = Sqr(a!*a! + b!*b! + c!*c!)
Compile Xelse
  Def Fn Diagonal!(a!, b!) = Sqr(a!*a! + b!*b!)
Compile End If

```

Note:

`Compile Long If` blocks can be nested to a depth of 16 levels.

See Also:

`BeginAssem; Tron;`

Compile ShutDown

statement✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Compile ShutDown "Optional Quoted Remark"
```

Description:

You may have some code that is specifically designed for one platform or another. For instance, 68K assembly cannot be compiled into a PPC application. In such cases, `Compile ShutDown` would abort the compile process with an error and show the offending line complete with the optional quoted remark.

Example

```
Compile Long If cpuPPC
    Compile ShutDown "This won't work in PPC."
Compile Xelse
    move.l    d0,-(SP)
Compile End If
```

See also:

```
Compile Long If; #If
```

Compile Xelse

statement

See the `Compile Long If` and `Compile End If` statements

CompileFlags

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
flag% = CompileFlags(_compileConstant)
```

Description:

Compile statements are directives to the compiler to change the way it evaluates and compiles code. (See `Compile` for a complete list of compiler directives.) Compile flags do not literally exist as accessible variables in the application but are used to set internal flags during compilation.

You can use this function to check specific flags which may be required by your application or to provide notes to another programmer about how to improve or streamline their code.

```
Long If CompileFlags(_dimmedVarsOnly) = _false
  Print "It is much better to check "Use only 1-
    Dimensioned variables" in the preferences Window."
End If
```

In addition to all of the standard compiler flags, this function can return additional information.

```
_FB3CpuCode      = 0x00020000      ' 0=68K/1=-PPC
_FB3ErrorCount   = 0x00040000      '# of errors so far
_FB3WarningCount = 0x00080000      '# of warnings so far
_FB3FloatDivChar = 0x00100000      '"/" or "\"
_FB3HideWarnings = 0x00200000      'show/hide warnings
```

See Also:

`Compile`; `Compile Long If`; `Compile ShutDown`

CompilerVersion

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
versionNum& = CompilerVersion
```

Description:

This function returns the current version of the compiler. The number is internally set before each compiler is shipped. The purpose of this function is to determine whether or not a specific functionality is present before a particular command is executed.

If, for example, you wanted to invoke a routine that was not added to the compiler until version 3 release 2, you would check the version as follows:

```
Long If CompilerVersion < 0x03020000
  Print "You need a newer compiler to run this."
Xelse
  Rem new operation goes here.
End If
```

The `CompilerVersion` function always returns a long word which is formatted as follows:

Byte 1 = Version number
 Byte 2 = Release number
 Byte 3 = Revision Number
 Byte 4 = Fix

Compound function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
compoundFactor# = Fn Compound#(rate, periods)
```

Description:

Returns the compounding factor for the given interest rate and number of periods. The interest rate should be expressed as a fraction of 1; for example, 5.2 percent should be expressed as 0.052.

Note:

Compound uses the following formula:

$$\text{compoundFactor\#} = (1 + \text{rate})^{\text{period}}$$

See Also:

Annuity

Compress Dynamic

statement✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**`Compress Dynamic arrayName`**Revision:**

May, 2001 (Release 5)

Description:

When space is required for a dynamic array, it is allocated in chunks. The size of these chunks is determined by the global variable `gFBDynamicGrowInc&`. If `gFBDynamicGrowInc&` is set to 1000 and you use a single element, 1000 elements are allocated. No additional elements are allocated unless and until you pass the 1000 mark. If you needed to access the 1001st element, an additional 1000 elements would be allocated.

The preallocation of space in large chunks can make dynamic arrays operate at a much faster pace because the handle used to accommodate the information is not resized as often.

When you want to regain RAM set aside for a dynamic array that has not yet been addressed, use `Compress Dynamic`. The size of the handle used to hold information is reduced. Internal counters that track the maximum number of elements available before resizing are also corrected.

See Also:`Dynamic, Read Dynamic, Write Dynamic`

Constant declaration**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
_constantName = staticExpr
```

Description:

This is a non-executable statement which assigns the value of *staticExpr* to the symbolic constant indicated by *_constantName*. The *_constantName* must indicate a name which was not defined anywhere previously in the program, and which is different from all pre-defined FB^3 symbolic constant names. The name must be preceded by an underscore (*_*) character. The *staticExpr* must be a “static integer expression”; it cannot contain any variables nor function references.

The constant declaration statement is one of several ways to assign values to symbolic constant names. Other ways include using the `Dim Record` statement and using the `Begin Enum` statement.

See Also:

`Dim Record`; `Begin Enum`; `Let`; *Appendix C: Data Types and Data Representation*

Coordinate Window

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

Coordinate Window

Description:

This statement does nothing, but is retained for backwards-compatibility with older versions of FutureBASIC.

Cos function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
theCosine# = Cos(expr)
```

Description:

Returns the cosine of *expr*, where *expr* is given in radians. The returned value will be in the range -1 to $+1$. Cos always returns a double-precision result.

Note:

To find the cosine of an angle *degAngle* which is given in degrees, use the following:

```
theCosine# = Cos(degAngle * pi# / 180)
```

where *pi#* equals 3.14159265359

See Also:

```
Acoss; Sin; Tan; Ussr _cosine
```

Cosh

function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

`result# = Cosh(expr)`

Description:

Returns the hyperbolic cosine of `expr`. `Cosh` always returns a double-precision result.

Note:

`Cosh(x)` is defined as: $\frac{e^x + e^{-x}}{2}$

See Also:

`Acosh`; `Sinh`; `Tanh`; `Exp`

CsrLin**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
currentLine = CsrLin
```

Description:

This function returns the number of the “current” text line (that is, the text line which contains the current pen position) for the current window. The text line at the top of the window is considered Line #0.

CsrLin does not necessarily reflect the number of text lines which have actually been displayed. It is calculated based on the current pen position and the size of the current font.

Example:

```
Window 1
Text _monaco, 16
Cls
For i = 1 To 10
  Print "CsrLin = "; CsrLin
Next
```

Console Behavior:

When you use the Console runtime, CsrLin switches to the Text Window before executing. The value returned is the actual line number in the edit field and remains accurate even when the field is scrolled.

See Also:

Pos; Window function

Cursor statement

✓ *Appearance*✓ *Standard*✗ *Console***Syntax (Standard BASIC):**

```
Cursor [=] intExpr
```

Syntax (Appearance Runtime):

```
Cursor cursorID[, cursorType]
```

Revision:






February, 2002 (Release 6)

Description:

If *intExpr* is positive, it's interpreted as the resource ID of a 'crsr' (first choice) or a 'CURS' resource (alternative choice), and this statement changes the current cursor to the indicated cursor. The following cursor resources are found in the System file, and are always available:

For Appearance Manager calls, *cursorType* must be `_themeCursorStatic` or `_themeCursorAnimate`

Standard BASIC:

<i>Resource ID</i>	<i>Value</i>	<i>Cursor</i>
<code>_arrowCursor</code>	0	
<code>_iBeamCursor</code>	1	
<code>_crossCursor</code>	2	
<code>_plusCursor</code>	3	
<code>_watchCursor</code>	4	

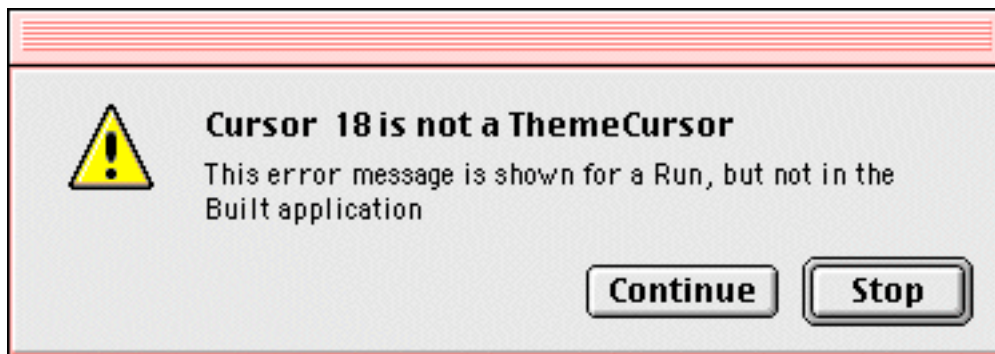
Appearance manager Cursors

Items marked with an asterisk (*) may be animated.

<code>_kThemeArrowCursor</code>	(0)	
<code>_kThemeCopyArrowCursor</code>	(1)	
<code>_kThemeAliasArrowCursor</code>	(2)	
<code>_kThemeContextualMenuArrowCursor</code>	(3)	
<code>_kThemeIBeamCursor</code>	(4)	
<code>_kThemeCrossCursor</code>	(5)	
<code>_kThemePlusCursor</code>	(6)	
<code>_kThemeWatchCursor</code>	(7) *	
<code>_kThemeClosedHandCursor</code>	(8)	
<code>_kThemeOpenHandCursor</code>	(9)	
<code>_kThemePointingHandCursor</code>	(10)	
<code>_kThemeCountingUpHandCursor</code>	(11) *	
<code>_kThemeCountingDownHandCursor</code>	(12) *	
<code>_kThemeCountingUpAndDownHandCursor</code>	(13) *	
<code>_kThemeSpinningCursor</code>	(14) *	
<code>_kThemeResizeLeftCursor</code>	(15)	
<code>_kThemeResizeRightCursor</code>	(16)	
<code>_kThemeResizeLeftRightCursor</code>	(17)	

Error Messages (Appearance Manager)

If you attempt to use an illegal value in the cursor statement, you will see the following dialog:



This type of dialog will only appear when you select Run from the Command menu. It will not show up if you use an illegal value in the program after selecting Build. This prevents the error message from appearing on your end user's screen, but still makes it available during the coding process.

Forcing a Cursor Event:

If `intExpr` is negative (for example: `Cursor = -_watchCursor`), then the cursor is changed to the one whose ID is `Abs(intExpr)`, and a cursor event is generated, which can be trapped by checking for events of type `_cursorEvent` with the `Dialog` function. You cannot generate a `_cursorEvent` when switching to the `_arrowCursor` (since its resource ID number has no negative!)

Note:

If you've designed a `csrs` or a `CURS` resource of your own which you want to make available to your program, do the following:

1. Assign a positive resource ID number to the resource (don't use any of the numbers in the above table. You should use an ID number of 128 or higher.)
2. Copy the resource to the resources file which you reference in your program's `Resources` statement.

You can then use the `Cursor` statement to activate your cursor within the program.

See Also:

`Dialog` function cursor events; `Def Cycle` for animating cursors.

Cvi**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
var& = Cvi(string$)
```

Description:

This function converts the bytes in *string\$* into an integer value which has the same internal bit-pattern that *string\$* has. If *string\$* consists of 4 or more bytes, only its first 4 bytes are considered. If *string\$* consists of 1, 2 or 3 bytes, then *Cvi(string\$)* returns an 8-bit, 16-bit or 24-bit integer, respectively. If *string\$* is a null string, then *Cvi(string\$)* returns zero.

This function is useful for finding the integer form of such things as file types, creator signatures and resource types. For example:

```
ft$ = "TEXT"
theType& = Cvi(ft$)
```

After executing the above, *theType&* is then suitable for passing to a Toolbox routine which requires a file-type parameter. *theType&* will also have the same value as the integer constant `_TEXT`.

The size (in bytes) of the value returned by *Cvi* depends on the length of *string\$*. It does not depend on the current setting of *DefStr* *Byte/Word/Long*. Therefore, if you want to assign the return value of *Cvi* to a short integer variable, you must make sure that *string\$* is not longer than 2 bytes; otherwise, you'll get an unexpected value in your short integer variable. Similarly, if you want to correctly assign *Cvi*'s return value to a byte variable, you should make sure that *string\$* is not longer than 1 byte.

The *Mki\$* function is the inverse of *Cvi*. Note, however, that the output of *Mki\$* does depend on the current setting of *DefStr* *Byte/Word/Long*.

Note:

If *string\$* is 1 byte long, then *Cvi(string\$)* returns the same value as *Asc(string\$)*.

See Also:

Asc; *Mki\$*; *DefStr* *Byte/Word/Long*

Data statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Data item1 [,item2 ...]
```

Description:

This statement is used to list data constants (numbers or strings, quoted or unquoted) to be accessed by the `Read` statement. Each item must be a numeric or string constant; string constants may be quoted or unquoted. The items are separated by commas. Leading spaces (between a comma and the item that follows it, or between the `Data` keyword and the first item) are ignored; therefore, if you wish to represent a string item which contains commas and/or leading spaces, you must enclose it in quotes. Trailing spaces in an unquoted string item are not ignored; they're considered part of the string.

To represent a numeric item in a `Data` statement, you can use a decimal, hex, octal or binary constant.

You can have as many `Data` statements in your program as you like, and as many or as few items in each `Data` statement as you like. The only restriction is that the total number of data items (in all `Data` statements) must be at least enough to satisfy all `Read` requests. You can use the `Restore` statement to allow `Data` items to be read more than once.

`Data` statements are global in scope: this means that any `Read` statement (whether it's in a local function, or in "main") can access any `Data` statement (whether it's in a (possibly different) local function, or in "main"). `Data` statements are "non-executable," which means you can't change their effect by putting them inside a conditional execution structure such as `Long If...End If`. However, you can conditionally include or exclude them from the program by putting them inside a `Compile Long If` block.

Note that everything between the `Data` keyword and the end of the line is considered part of the `Data` statement. In particular, this means that you cannot use the ":" separator to put another statement after the `Data` statement on the same line, and you cannot put a comment after the `Data` statement on the same line.

See Also:

`Read`; `Restore`; `Compile Long If`

Date\$

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
dateString$ = Date$
```

Description:

The `Date$` function reads the system clock and returns the current date as a string in the format: MM/DD/YY (that is, the string returned contains two digit numerals each for month, day and year, separated by slash marks).

You can use the Toolbox routine `DateString` to get a date string that is formatted according to the international specifications set in the “Date & Time” control panel. See the sample program on the CD to see how this is done.

Example:



CD Example: Time & Date formats.bas

Note:

The `Date$` function only returns the last two digits of the year. To get the complete 4-digit year value, you can use the `SecondsToDate` Toolbox function, as follows:

```
Dim dateRec As DateTimeRec
Dim secs&
Dim theMonth%, theDay%, theYear%, theWeekday%
Call GetDateTime( secs& )
Call SecondsToDate(secs&, dateRec)
theMonth%   = dateRec.month
theDay%     = dateRec.day
theYear%    = dateRec.year      'Full 4 digits
theWeekday% = dateRec.dayOfWeek '1=Sunday, 7=Saturday
Print theMonth%
Print theDay%
Print theYear%
Print theWeekday%
```

See Also:

`Time$; Timer`

Dec statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Dec(intVar)
numericVariable -= valueToSubtract
```

Revision:

May 30, 2000 (Release 3)

Description:

This statement decrements *intVar* by 1; that is, it subtracts 1 from the value of *intVar*, and stores the value back into *intVar*. *intVar* must be a (signed or unsigned) byte variable, short-integer variable or long-integer variable. If *intVar* is already at the minimum value for its variable type, then `Dec(intVar)` will cycle it back to its maximum value. As of Release 3, FB^3 supports the use of `-=` to decrease the value of a variable by a specified amount.

Example:

```
Dec(x&)
```

and...

```
x& -= 1
```

...is equivalent to:

```
x& = x& - 1
```

... or:

```
x&--
```

The following expressions are also equivalent.

```
x& = x& - 100
x& -= 100
```

Note

The `-=` syntax may not be used for arrays of strings, containers, or records. Where arrays are involved, only numeric values may take advantage of this syntax.

See Also:

```
Inc; Dec Long/Word/Byte; Def Cycle
```

Dec Long/Word/Byte**statements**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:****Dec** {**Long**|**Word**|**Byte**} (*addr&*)**Description:**

This statement decrements the long integer, short integer or byte which begins at the specified address in memory; that is, it subtracts 1 from the value in memory and stores the result back into the addressed location. If the long integer, short integer or byte is already at its minimum possible value, then the statement will cycle it back to its maximum value.

Example:**Dec Long** (*myAddr&*)

...is equivalent to:

Poke Long *myAddr&*, **Peek Long**(*myAddr&*) - 1

Also:

Dec Word (*myAddr&*)

...is equivalent to:

Poke Word *myAddr&*, **Peek Word**(*myAddr&*) - 1**See Also:****Dec**; **Inc**; **Inc Long/Word/Byte**

Def ApndLng**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def ApndLng(longExpr, Handle&)
```

Description:

Appends the long-integer expression given by *longExpr* to the end of the relocatable block specified by *Handle&*. This increases the size of the block by 4 bytes.

Note:

This statement could cause a system error if heap memory is very low or very fragmented. You can use the `Mem(_maxAvail)` function periodically to de-fragment memory.

See Also:

Def ClearHandle; Def DisposeH; Mem, Inside Macintosh: *Memory*

Def ApndStr**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def ApndStr(string$, STR#resourceHndl&)
```

Description:

This statement adds the specified *string\$* to the end of the relocatable block specified by *STR#resourceHndl&*. *STR#resourceHndl&* should be a handle to an existing “STR#” resource, or a handle to a block which you intend to save as an STR# resource. To create a new, empty handle for use with `Def ApndStr`, you should create it like this:

```
myHandle& = Fn NewHandle _clear (2)
```

After you have added strings to the new handle, you can save it as a resource by calling the `AddResource` routine.

To get a handle to an existing “STR#” resource for use with `Def ApndStr`, you can use any of a variety of Resource Manager functions such as `GetResource`.

Note:

Never use `Def ApndStr` on a purgeable resource, unless you first call `Fn HNoPurge(str#Handle&)` to (temporarily) make it un purgeable. If you think the resource may have been purged before you had a chance to call `Fn HNoPurge`, then you should also call `LoadResource(str#Handle&)` to make sure it's loaded into memory.

If you are using `Def ApndStr` to update an existing “STR#” resource, then use `Call ChangedResource(str#Handle&)` after using `Def ApndStr`, to cause your changes to be written to disk when the resource file is closed or updated.

`Def ApndStr` could cause a system error if heap memory is very low or very fragmented. You can use the `Mem(_maxAvail)` function periodically to de-fragment memory.

See Also:

Str#; `Def RemoveStr`; “Resource Manager” chapter in Inside Macintosh: *More Macintosh Toolbox*.

Def BlockFill & Def LongBlockFill**statements**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

Def BlockFill(startAddr&, numBytes&, byteValueExpr)
Def LongBlockFill(startAddr&, numBytes&, byteValueExpr)

```

Revision:

July 26, 2000 (Release 3)

Description:

Fills each byte in a range of memory with the value specified in *byteValExpr*. The *startAddr&* parameter indicates the first memory address to fill, and *numBytes&* indicates the number of bytes in the range. **Def BlockFill** and **Def LongBlockFill** are identical.

Example:

CD Example: Def BlockFill.bas

See Also:

BlockMove; String\$; Space\$

Def Border**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def Border(twoPixelBorder, {rect | #rectAddr&})
```

Description:

Draws a double border (in the current color and pattern) around the outside of the rectangle which is specified by *rect* (which should be an 8-byte variable such as a `Rect` type), or pointed to by *rectAddr*&. If *twoPixelBorder* is “true” (nonzero), then the outer border is drawn with a 2-pixel pen width; otherwise it's drawn with a 1-pixel pen width.

Example:

CD Example: Def Border.bas

Console Behavior:

When you use the Console runtime, `Def Border` switches to the Graphics Window before executing.

See Also:

```
Def TitleRect; Def ShadowBox; Def <just>Box
```


Def BtnRect statement

✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
Def BtnRect({rect | #rectAddr&}, buttonID%)
```

Description:

If *buttonID%* is the ID number of a button or scroll bar in the current window, then this statement returns the button's (or scroll bar's) bounding rectangle into *rect* (which should be an 8-byte variable such as a `Rect` type), or into the 8 bytes starting at address *rectAddr&*. A button's ID number is assigned by the `Button` statement; a scroll bar's ID number is assigned by the `Scroll Button` statement.

Example:

CD Example: Def BtnRect.bas

Note:

If the specified button does not exist, then the rectangle (0,0)-(0,0) is returned.

See Also:

`Button`; `Button&`; `Dialog`

Def ButtonHelpSetText**statement**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
Def ButtonHelpSetText( btnID%|&, helpMessage$)
```

Revision:

May, 2003 (Release 8)

Description:

This routine associates the *helpMessage\$* string with the button specified by *btnID*.

Behavior for Carbon applications:

Help tags are meant to replace Help balloons in Carbon. They should provide concise and precise information about the action of the button located under the mouse pointer. Help tags are displayed automatically by the system when no event of a higher priority is pending and they disappear from the screen as soon as an input from the user must be handled.

Because Help tags are much more subtle than the outdated Help balloons their visibility status defaults to on. You can toggle the display of the Help tags with the routine `Def ButtonHelpShow(visibilityFlag)`.

The *helpMessage\$* string parameter may contain escape codes to setup the alignment of the Help tag displayed on screen and the extra message that usually appears when the Command key is held down. The string might be formatted like so:

```
"helpmessage[\$[+]additionalMessage][@alignmentCode]"
```

where:

`\$` indicates the ending of the first help message and the beginning of the second. Optionally, you can add the second message to the first using the + symbol.

`@` *alignmentCode* indicates the position of the Help tag relative to the button specified. You can use one of the following alignment codes (T, L, B, R and C stand for top, left, bottom, right and center respectively:

```
T; L; B; R; LC(CL); RC(CR); TL; TR; LT; LB; BL; BR; RT; RB.
```

For example, the string below would display an Help tag at the top of the button. Holding down the Command key would append a carriage return + more info.

```
message = "Close the demo\$+and I could have a rest\@T"
```

Behavior for PPC (starting with Mac OS 8.5)

PPC applications use the Help balloons mechanism to display the message associated with a given button. However, an extra step is required to make the balloon show up effectively on screen. When your dialog handler intercepts a cursor event, you must call `Def ButtonHelpDisplay` if the mouse is over a button located in the active window. If Help balloons are enabled at the system level then the help message is displayed. Note that the second help message is ignored in PPC.

Note:

You must include “Subs Help tags.Incl” in your project to make this command available to your program.

Example:

```
Def ButtonHelpSetText (_quitBtn, "Shutdown the application")
```

**See Also:**

```
Def ButtonHelpShow; Def ButtonHelpDisplay (PPC)
```

Def ButtonHelpShow**statement**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**`Def ButtonHelpShow(showHideFlag)`**Revision:**

October, 2002 (Release 8)

Description:

Because Help tags are indeed less intrusive than Help balloons that they replace in Carbon, Apple has decided that their default behavior is popping up on screen as time permits. On the other hand, Help balloons are turned off by default. However, you or your end-user may wish to act upon that behavior.

`Def ButtonHelpShow` lets you control the display of Help tags in Carbon or Help balloons in PPC. If *showHideFlag* is zero the Help messages are no longer displayed, any other value for this parameter will cause the reappearance of the Help tags on screen, as soon as the system determines it is time to do so.

An additional step is required for non Carbon applications in order to display the Help balloons associated with buttons created in FutureBASIC: you must call `Def ButtonHelpDisplay` at an appropriate time, i.e. when the mouse enters the button area.

Note:

You must include “Subs Help tags.Incl” in your project to make this command available to your program.

See Also:`Def ButtonHelpSetText; Def ButtonHelpDisplay`

Def ButtonHelpDisplay**statement**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
Def ButtonHelpDisplay (btnID%| &)
```

Revision:

May, 2003 (Release 8)

Description:

In non Carbon applications, Help balloons are used to display the help message associated with a given button. Unlike Help tags in Carbon, Help balloons related to FutureBASIC buttons will not show up magically on screen. Your application must determine when the Help balloon must be displayed. You will generally intercept a `_cursorOver` event in your Dialog handler to retrieve the `btnID` over which the mouse pointer is located. Help balloons must be enabled at the system level in order to see effectively the Help balloons on screen.

This command is ignored in Carbon applications.

Note:

You must include “Subs Help tags.Incl” in your project to make this command available to your program.

See Also:

```
Def ButtonHelpSetText; Def ButtonHelpShow
```

Def CBox

statement

See the Def <just>Box statements.

Def ChangedResource**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:****Def ChangedResource** (*rHndl*&)**Revision:**

August, 2002 (Release 7)

Description:

There are probably more potential problems in dealing with the Resource Manager than any other part of the Mac toolbox. One bug in the manager's code is that every time a resource is marked as changed, space is reserved on the disk for a new version without removing any previous requests. So if you change a resource 100 times, then space will be reserved on your hard drive for 100 copies. This becomes a problem because the file size and the number of resources allowed in a single file is finite.

This procedure will only mark a handle as changed (a) it is a resource handle and (b) it is not already flagged as a changed resource.

See Also:

Usr ReplaceResource; Usr OpenRFPerm

Def CheckOneItem**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def CheckOneItem(menuID%, itemID%)
```

Description:

Places a checkmark next to the specified item in the specified menu, and unchecks all other items in that menu. The menu ID number is assigned by the `Menu` statement, or by a Toolbox routine that creates the menu, or by the resource that defines the menu. The item ID corresponds to the item's position in the menu (the item just under the menu title is item #1). Note that gray "dividing lines" in menus are also counted as menu items.

Use an `itemID%` value of zero to uncheck all items in the menu.

Example:

CD Example: Def CheckOneItem.bas

Note:

Do not set `itemID%` to a menu item which has a hierarchical submenu attached to it.

See Also:

`Menu`; `SetItemMark` Toolbox routine

Def ClearHandle

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Def ClearHandle(Handle&)
```

Description:

If *Handle&* is a valid handle to a relocatable memory block, this statement sets all of the bytes in the block to zero. If *Handle&* is not a valid handle, the statement does nothing.

See Also:

```
Def DisposeH; Inside Macintosh: Memory
```

Def CreateResFile**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def CreateResFile(fileName$)
```

Revision:

August, 2002 (Release 7)

Description:

Def CreateResFile replaces the obsolete Call CreateResFile which has seen such heavy use in the past. Mac OS X made the call obsolete, but by exchanging the Def keyword for the optional Call keyword, you can breath new life into your programs. The Def version of the call works in all versions of the system software from system 7.x to Mac OS X. If this call is successful, a resource file is created in the current directory. You may set the current directory using Folder.

Def CreateResFile may also be used to add a resource fork to an existing file. Creating the resource file does not automatically open it.

See Also:

```
Def Open; Open; Usr OpenRFPPerm
```

Def Cycle	statement	
✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>

Syntax:

```
Def Cycle (min%, max%, var%)
```

Description:

Use this statement when you need to cycle the value of an integer variable through a series of consecutive values and then back to the minimum value. The `Def Cycle` statement is equivalent to the following lines:

```
Inc (var%)  
If var% < min% Or var% > max% Then var% = min%
```

min% and *max%* must be short integer values, and *var%* must be a short integer variable.

Def DebugNumber

statement

✓ *Appearance* ✓ *Standard* ✓ *Console*

Syntax:

```
Def DebugNumber (numericExpression)
```

Revision:

May, 2001 (Release 5)

Description:

This statement opens a small window in the bottom right corner of the screen and displays a numeric value in decimal, hexadecimal, and binary. It waits for your mouse click, then goes away.

Def `DebugNumber` is a perfect solution for programmers needing quick information who do not wish to go through the process of setting up the debugger and do not have the knowledge required to use `MacsBug`.

```
Dim x As Int
Dim y As Double
x = 100
y = 1.234
Def DebugNumber(x * y)
```

Program output:

Decimal:	123.399993896
Hex;	0000007B
Binary	00000000000000000000000001111011

Click mouse to continue.

See Also:

```
Def DebugString
```

Def DebugString**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def DebugString (stringExpression)
```

Revision:

May, 2001 (Release 5)

Description:

This statement opens a small window in the bottom right corner of the screen and displays a string. It waits for your mouse click, then goes away.

Def DebugString is a perfect solution for programmers needing quick information who do not wish to go through the process of setting up the debugger and do not have the knowledge required to use MacsBug

```
Local Fn fred
  Def DebugString("The program made it into Fn fred")
End Fn

Fn fred
```

Program output:

The program made it into Fn fred

Click mouse to continue.

See Also:

Def DebugNumber

Def DisposeH**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:****Def DisposeH**(*Handle&*)**Description:**

If *Handle&* represents a valid handle to a relocatable memory block, this statement disposes of the block, and sets the value of *Handle&* to zero (*Handle&* must be a long-integer variable or a *Handle* variable).

If *Handle&* does not represent a valid handle, the statement sets the value of *Handle&* to zero, but otherwise does nothing.

Note:

Never use `Def DisposeH` on a resource.

See Also:

`Def ClearHandle`; `Kill Field`; `DisposeHandle` Toolbox routine; Inside Macintosh: *Memory*

Def DrawImageFile

statement✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
Def DrawImageFile ( fName$ , vRef% )
```

Revision:

May, 2001 (Release 5)

Description:

You may use this statement to display a picture directly from disk. The picture can actually be too large to load into memory. You may determine the size of the picture in advance using Def GetImageFileRect. The following example shows how to open a file and display it in a window.

Note:

In order to use this routine, you must include the file named "Subs Image Files.Incl" as shown in the example below.

```
Include "Subs Image Files.Incl"

Dim fName$
Dim @vRef%
Dim r As Rect
gFBUseNavServices = _ztrue
fName$ = Files$(_fOpenPreview,, "Select a file to Open", vRef%)

Long If fName$[0]
  Def GetImageFileRect(fName$, vRef%, r)
    OffsetRect(r, -r.left, -r.Top)
    Window 1, fName$, @r, _docNoGrow
    Def DrawImageFile(fName$, vRef%)
  End If

  Print @(1,1) "Click to exit"

Do
Until Fn Button
```

See Also:

Def GetImageFileRect

Def EmbedButton**statement**✓ *Appearance*✗ *Standard*✗ *Console***Syntax:**

```
Def EmbedButton(childButtonID&, parentButtonID&)
```

Revision:

February, 2002 (Release 6)

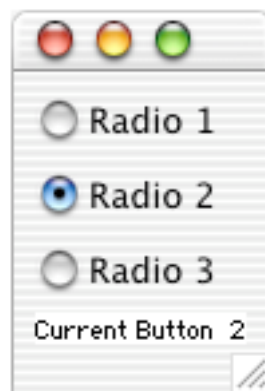
Description:

The appearance manager can embed several buttons (sub controls) within a single parent button (or super control). In fact, buttons may be embedded in buttons which are in turn embedded in other buttons. The advantages are enormous. Take this example: each window has a single root control. All buttons within the window are embedded into this root control or into one of its children. When the window becomes inactive, the root control is disabled. With that single command (implemented by the FB runtime) all other controls in the window are automatically disabled.

The versatility extends to things like showing or hiding a pane in a tab control or buttons within a group.

Example:

The following example creates a radio button group and embeds individual buttons into the super control. As a result, you can poll the group button to find out which of the radio sub controls is currently selected.




```

Dim r      As Rect
Dim pR     As Rect
Dim h      As Handle
Dim bRef   As Long
Dim err    As OSErr

// setup
_btnHt     = 20
_btnWd     = 80
_btnMargin = 8
bRef       = 1

// create a window
SetRect(r,0,0,_btnWd_btnMargin_btnMargin,120)

Appearance Window 1,,@r

err = Fn SetThemeWindowBackground( Window( _wndPointer ),-
    _kThemeActiveDialogBackgroundBrush, _zTrue )

// button #1 is the papa button
// note that the parent button has sufficient space so that
// it holds all embedded buttons within its own rectangle

SetRect(r,_btnMargin,_btnMargin,_btnMargin_btnWd,(_btnMargin_btnHt)*3)
Appearance Button bRef,_activeBtn,0,0,1,-
    "",@r,_kControlRadioGroupProc

bRef ++
SetRect(r,_btnMargin,_btnMargin,_btnMargin_btnWd,_btnMargin_btnHt)
Appearance Button bRef,_activeBtn,0,0,1,-
    "Radio 1",@r,_kControlRadioButtonProc
Def EmbedButton(bRef,1)

bRef ++ : offsetrect(r,0,_btnHt_btnMargin)
Appearance Button bRef,_activeBtn,0,0,1,-
    "Radio 2",@r,_kControlRadioButtonProc
Def EmbedButton(bRef,1)

bRef ++ : offsetrect(r,0,_btnHt_btnMargin)
Appearance Button bRef,_activeBtn,0,0,1,-
    "Radio 3",@r,_kControlRadioButtonProc
Def EmbedButton(bRef,1)

Local Fn HandleDialog
    Dim As Long action,reference
    action = Dialog(0)
    reference = Dialog(action)
    Long If action = _btnClick
        moveTo(8,100)
        Print "Current Button ";Button(1);
    End If
End Fn

On Dialog Fn HandleDialog

Do
    HandleEvents
Until 0

```

Def Flash

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

Def Flash

Revision:

July 26, 2000 (Release 3)

Description:

Inverts the current output window. A subsequent call to Def Flash returns the window to normal. You can use Def Flash to draw attention to a window through repeated calls or as a debugging tool to determine which window is currently used for output.

Def Fn <expr>**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def Fn functionName [(var1 [, var2 ...])] = expr
```

Description:

This statement defines a “one-line” function. You can refer to the function in later parts of your program by using an expression in this form:

```
Fn functionName [(parm1 [, parm2 ...])]
```

This expression returns the value of `expr` from the function definition.

The `Def Fn <expr>` statement should not appear inside any `Local` function.

`functionName` can be any valid FB^3 identifier which is different from any other function name defined in your program. `functionName` can optionally end with a type-identifier suffix (such as %, \$, &, etc.)

If `functionName` ends with a type-identifier suffix, the suffix indicates the data type that the function returns (and hence `expr` should be of the same type). If no type-identifier suffix is specified, the function returns a long-integer value.

You can optionally include a formal parameter list in the function definition: this is a list of variable names (`var1`, `var2`, etc.) separated by commas and enclosed in parentheses, which immediately follows `functionName`. Usually, `expr` will contain references to these parameter variables. When you call a function that has a formal parameter list, you pass values to it in an actual parameter list (`parm1`, `parm2`, etc.). These values are then assigned to `var1`, `var2`, etc., and are used in evaluating `expr`.

`var1`, `var2`, etc. must be “simple” variables: they cannot be array elements, records, nor record fields. `parm1`, `parm2`, etc. can (with some exceptions) be any kinds of expressions, as long as the data type of each `parm` expression is compatible with its corresponding `var` variable in the formal parameter list. The number and order of the items in the actual parameter list must exactly match the number and order of the items in the formal parameter list (if any).

The variables in the formal parameter list are either global (if they were previously declared within a `Begin Globals...End Globals` section), or they are “local to main.” In either case, this means that the values which get assigned to those variables (when you call the function) persist even after the function returns its value. You need to keep this in mind if you later execute some statement in “main” (outside of all `Local` functions) which contains one of those variables.

`expr` may contain other variables besides those which appear in the formal parameter list. All variables in `expr` are either global (as declared within a `Begin Globals...End Globals` block) or are local to main.

Example:

```

Def Fn Area!(r!) = pi# * r! * r!
:
Local Fn Circle6
  a! = Fn Area!(6.0)
  Print a!
End Fn

```

The function `Fn Area!` calculates the area of a circle, when the radius of the circle is passed as a parameter. We are assuming that the variable `pi#` is a global variable or a “local to main” variable whose value has previously been set to 3.14159...

When we call the `Circle6` function, the value 113.079 gets assigned to the local variable `a!`. As a side effect of calling `Fn Area!(6.0)`, the value of the “local to main” variable `r!` is changed to 6.0.

Note:

`Def Fn <expr>` is a “non-executable” statement, which means you cannot affect the definition of the function by placing `Def Fn <expr>` after `Then` or `Else` (in an `If` statement), nor by placing it inside any kind of “conditional execution” block such as `Long If...End If`, `While...Wend`, `For...Next`, etc. However, you can affect the function definition (at compile time) by placing `Def Fn <expr>` inside a `Compile Long If` block.

`Def Fn` does not work for threaded functions.

See Also:

```

Local Fn; End Fn; Fn Using; Fn; @Fn; Def Fn Using <Fn address>

```

Def Fn <prototype> statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Def Fn functionName [(var1 [,var2 ...])]
```

Description:

This statement declares a prototype for a `Local Fn`. The *functionName* and the argument list (*var1*, *var2* etc.) must match those of some `Local Fn` whose definition appears later in the source code stream.

A `Local Fn` must either be defined (using a `Local Fn...End Fn` block) or prototyped (using `Def Fn <prototype>`) before it can be referenced in any `Fn <userFunction>` statement. By prototyping the function early in the code, you can call `Fn <userFunction>` above the spot where the `Local Fn...End Fn` block actually appears. This frees you from concerns about how to order your `Local Fn` blocks in the code.

Example:

In the following excerpt, `Fn myFn1` calls `Fn myFn2`, and `Fn myFn2` calls `Fn myFn1`. This kind of construction would be difficult to implement without prototyping the functions.

```
Def Fn myFn1(x As Long)
Def Fn myFn2(x As Long)

Do
  Input "Enter a number", k
  Print Fn myFn1(k)
Until k = 0

Local Fn myFn1(x As Long)
  If x Mod 1 Then z = 3 * x Else z = Fn myFn2(x) + 6
End Fn = z

Local Fn myFn2(x As Long)
  If x Mod 1 Then z = Fn myFn1(x) - 1 Else z = x / 2
End Fn = z
```

See Also:

`Fn <userFunction>`

Def Fn Using <Fn address> statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Def Fn functionName [(var1 [,var2 ...])] Using FNaddress&
```

Description:

This statement associates the name *functionName* with the routine which is located at the address given by *FNaddress&*. You can refer to *functionName* in later parts of your program by using an expression in this form:

```
Fn functionName [(parm1 [, parm2 ...])]
```

This expression will call the function referenced by *FNaddress&*, and will return the value (if any) that the referenced function returns.

The `Def Fn Using <Fn address>` statement should not appear inside any `Local` function.

FNaddress& must be a long-integer variable or a `Pointer` variable. Before you can call `Fn functionName`, you must make sure that the *FNaddress&* variable contains the address of a `Local Fn`, a `Long Fn` or a `Def Fn <expr>`, as returned by the `@Fn` function. You must not use the address of a label location (as returned by the `Line` function or the `Proc` function).

If the name of the function referenced by *FNaddress&* ends with a type-identifier suffix, then *functionName* must end with the same type-identifier suffix. If the function referenced by *FNaddress&* has a parameter list, then you must include a parameter list (*var1 [,var2...]*) in the `Def Fn Using <Fn address>` statement. The number, order, and data types of the parameters in the `Def Fn Using <Fn address>` list must match the number, order, and data types in the parameter list of the referenced function.

The `Def Fn Using <Fn address>` statement is useful in cases where your program must decide at runtime which of several similar functions should be executed in a given instance.

Note:

`Def Fn Using <Fn address>` is a “non-executable” statement, which means you cannot change its effect by placing it after `Then` or `Else` (in an `If` statement), nor by placing it inside any kind of “conditional execution” block such as `Long If...End If`, `While...Wend`, `For...Next`, etc. However, you can conditionally include or exclude it by placing it inside a `Compile Long If` block.

It is possible to choose at run time which function *functionName* shall be associated with, and even to dynamically change that association from one function to another. This is done by dynamically setting the *FNaddress&* variable to the addresses of different functions at run time.

See Also:

```
Local Fn; End Fn; @Fn; Def Fn <proTotype>
```

Def GetButtonData**statement**✓ *Appearance*✗ *Standard*✗ *Console***Syntax:**

```
Def GetButtonData (bRefNum&, part%, tagType&,  $\neg$ 
                  maxSize&, dataPtr& , actualSize&)
```

Revision:

February, 2002 (Release 6)

Description:

Def GetButtonData is a mechanism that may be used to extract specific information from a control. In the past, the only information needed for a control was its current value. For instance, a check box was either checked or it wasn't. Appearance Manager controls contain additional information that varies from one type of button to the next. The specific information required by this procedure depend on the type of control and the type of data required by your application.

Def GetButtonData is the compliment of Def SetButtonData which is documented on page 170. The parameters for these two calls are identical with two exceptions:

maxSize& the maximum size of data for which your program has allocated space

actualSize& after the call is completed, this variable will contain the actual length of the data retrieved.

You may determine the size of available data in advance by using a runtime function:

```
actualDataSize& = Fn ButtonDataSize( bRefNum&, part%, tagType& )
```

See Also:

Def SetButtonData, Button function, Appearance Button

Def GetButtonTextSelection**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def GetButtonTextSelection(bRefNum&, selStart, selEnd)
```

Revision:

February, 2002 (Release 6)

Description:

Use this procedure to get the selection range of an Appearance Manager, control-based edit field. (Such buttons are created using the `Appearance Button` statement with a type like `_kControlEditTextProc.`) The minimum value for *selStart* is zero for the position before the first character. The maximum value for *selEnd* is 32767 (or `_maxInt`).

Because this routine fills in the variables passed as parameters, you must pass integer variables to get the desired results. The fragment that follows shows one possible method for doing this.

```
Dim selStart As Word
Dim selEnd   As word
Def GetButtonTextSelection( bRefNum&, selStart, selEnd)
```

See Also:

`Appearance Button`; `Def SetButtonData`; `Button function`,
`Def SetButtonTextSelection`

Def GetImageFileRect**statement**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
Def GetImageFileRect (fName$, vRef%, rectangle)
```

Revision:

May, 2001 (Release 5)

Description:

You may use this statement to determine what rectangle may be used to display a picture. The rectangle is always repositioned so that its top left coordinates are zero.

```
Dim r As Rect
Def GetImageFileRect (fName$, vRef%, r)
```

Note:

In order to use this routine, you must include the file named "Subs Image Files.Incl" as shown in the example listed under `Def DrawImageFile`.

See Also:

`Def DrawImageFile`

Def <just> Box**statements**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def {CBox|LBox|RBox} ({rect | #rectAddr&}, stringVar$)
```

Description:

Draws the text in *stringVar\$* (using the current font, size, style, etc.) within the rectangle specified by *rect* (which should be an 8-byte variable such as a `Rect` type) or pointed to by *rectAddr&*.

No frame is drawn around the rectangle. If the rectangle is not wide enough to contain the entire string, then the string is automatically wrapped around to the next line, at word boundaries.

Each line of text is either `Centered`, `Left-justified` or `Right-justified` within the rectangle, depending on which variant of the statement you use.

No text is ever drawn outside of the rectangle; if the rectangle is not big enough to contain the entire string, then the text is clipped.

Example:

CD Example: Def jBox.bas

Console Behavior:

When you use the Console runtime, the `Def <just> Box` statements switch to the Graphics Window before executing.

Note:

To draw rectangle-bounded text that will redraw itself automatically (during a window refresh), use a static edit field (see the `Edit Field` statement).

See Also:

`Edit Field`

Def LBox

statement

See the Def <just> Box statements.

Def LCase**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def LCase(string$)
```

Revision:

July 26, 2000 (Release 3)

Description:

Converts all characters in the specified string to lower case. The original string is converted, so if you need to retain the characters in their initial form it is necessary to copy the string to a temporary variable before calling `Def LCase`. `Def LCase` does not work for containers.

Example:

```
Dim a$
a$ = "This is a TEST of Def LCase"
Print "Original: ";a$
Def LCase(a$)
Print "LCASEd   :";a$
```

Program output:

```
Original:This is a TEST of Def LCase
LCASEd   :this is a test of def lcase
```

See Also:

UCase\$

Def LCopy	statement	
✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>

Syntax:

Def LCopy

Revision:

July 26, 2000 (Release 3)

Description:

Sends a bit-mapped copy of the visible portion of the current output window to the printer.

Def Len**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def Len [=] stringVarLength
```

Description:

Use the `Def Len` statement to affect the amount of storage space that the compiler allocates to string variables (and string array elements and string record fields). This can help you to budget your memory more efficiently.

When the compiler first encounters a given string variable, it allocates enough storage space for the string's maximum length. The string's maximum length is determined as follows:

- If a length is explicitly declared in a `Dim` statement (as in: "`Dim 16 myString$`"), then that value is used.
- If the string is declared in a `Dim` statement using the `As Str255` clause, its maximum length is 255.
- If the maximum length is not explicitly declared in a `Dim` statement, then the `stringVarLength` specified in the most recent `Def Len` statement is used.
- If no `Def Len` statement has yet been encountered (and the maximum length is not explicitly declared in a `Dim` statement), then 255 is used.

(The actual amount of storage space allocated for the string is always at least 1 byte greater than the string's maximum length, and is always an even number of bytes.)

`stringVarLength` must have a value in the range 1 through 255, and must be a "static integer expression." A static integer expression is any valid expression which consists of only:

- integer literal constants;
- previously-defined symbolic constant names;
- operators (like `+`, `-`, `*`, `/`, `>`, `=`);
- parentheses

(In particular, it can't contain variables, nor function references.)

You can have more than one `Def Len` statement in your program. Each `Def Len` statement applies to the variables that are declared below it, until the next `Def Len` statement appears.

`Def Len` is global in scope: it applies to all variables that appear below it in the source code (until the next `Def Len` is encountered), whether those variables are within the "main" part of the program or within `Local` functions.

Example:

```

x$ = "Hello"
Def Len 35
Dim y$
Dim 255 z$
a$ = "This is a test"
Def Len 1
Dim b$(100)
Begin Record myRecType
    Dim 20 field1$
    Dim field2$
End Record

```

The compiler allocates storage space for the above strings as follows:

<i>Variable</i>	<i>Max String length</i>	<i>Bytes of storage</i>
x\$	255	256
y\$	35	36
z\$	255	256
a\$	35	36
b\$()	1 (for each element)	2 (for each element)
field1\$	20	22 (when a record variable is declared)
field2\$	1	2 (when a record variable is declared)

Note:

`Def Len` is a “non-executable” statement, which means you cannot change its effect by placing it after `Then` or `Else` (in an `If` statement), nor by placing it inside any kind of “conditional execution” block such as `Long If...End If`, `While...Wend`, `For...Next`, etc. However, you can conditionally include or exclude it by placing it inside a `Compile Long If` block.

See Also:

`Dim`; `Compile Long If`

Def LongBlockFill

statement

✓ Appearance	✓ Standard	✓ Console
--------------	------------	-----------

Syntax:

```
Def LongBlockFill(startPtr®, numBytes®, byteValExpr)
```

Description:

Fills each byte in a range of memory with the value specified in *byteValExpr*. The *startAddr*® parameter indicates the first memory address to fill, and *numBytes*® indicates the number of bytes in the range. Def LongBlockFill and Def BlockFill are identical.

See Also:

```
Def BlockFill
```


Def LPrint**statement**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**`Def LPrint`**Description:**

This statement initializes the printer driver and displays the printer “Job Dialog.” The exact appearance of the Job Dialog depends on which printer is currently selected, but it typically prompts the user to enter a range of page numbers to be printed, along with other information. Your program should execute the `Def LPrint` statement when the user clicks a “Print...” button or selects a “Print...” menu item in your program.

Note:

After your program executes `Def LPrint`, you can use the `PrCancel` function to determine whether the user cancelled the print job. If the user did not cancel, then you can use the `PrHandle` function to determine the page range and the number of copies that the user requested.

See Also:

`Clear LPrint`; `Close LPrint`; `LPrint`, `Def Page`, `PrHandle`

Def NewWindowPositionMethod**statement**✓ *Appearance*✗ *Standard*✗ *Console***Syntax:**

```
Def NewWindowPositionMethod(pMethod)
```

Revision:

February, 2002 (Release 6)

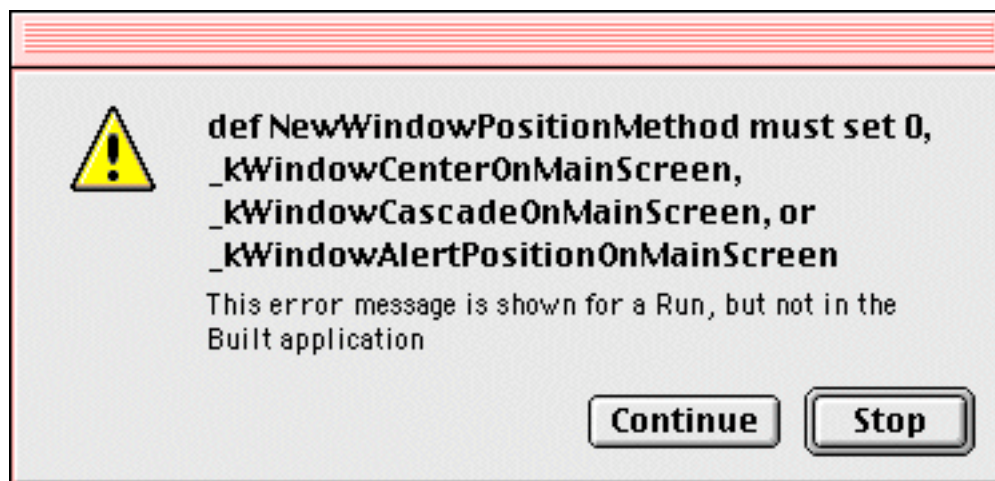
Description:

In the Appearance Manager you may determine, in advance, what positioning method will be used when a new window is created. This setting remains valid for all window created until it is set to zero by your code.

There are four valid *pMethod* parameters for NewWindowPositionMethod:

<i>pMethod</i>	<i>Action at next window built</i>
0	Do nothing.
_kWindowCenterOnMainScreen	Center window on the main monitor.
_kWindowCascadeOnMainScreen	Offset each new window on the main monitor as it is created
_kWindowAlertPositionOnMainScreen	Place the window just above the center point on the main monitor.

If you use any other parameters, you will see an alert when your program runs:

**See Also:**

Appearance Window; Def WindowReposition; Def TransitionRect

Def Open

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Def Open [=] typeCreaTorString$
```

Description:

Use this statement to specify the file type and/or creator signature to be assigned to files which are subsequently opened for output by the `Open` statement. *typeCreaTorString\$* should be a string expression whose length is either 4 or 8 characters. The first 4 characters will be used as the file type for newly opened files; the second group of 4 characters (if any) will be used as the creator signature for newly opened files. The specified *typeCreaTorString\$* remains in effect until another `Def Open` statement is executed.

Every Macintosh file has a 4-character file type and a 4-character creator signature. The file type is usually used to signify the general format of the file's contents. If you're creating a file which is to be opened by another application, you should use a file type which the other application recognizes. Otherwise, you can make up a custom file type for your program's private use. By default (i.e., if your program hasn't yet executed a `Def Open` statement), FB³ assigns the file type "?????" to files opened for output.

A file's creator signature usually signifies which application created the file. The Finder uses the file's creator signature to determine such things as: which application to launch when the user double-clicks the file's icon. The Finder uses the file's creator signature, in combination with the file's type, to determine the icon to display for the file. By default, if your program hasn't yet executed a `Def Open` statement with an 8-character *typeCreaTorString\$* parameter, FB³ assigns the creator signature "?????" to files opened for output.

Example:

```
Def Open "ttrotxt"
Open "O", #1, "test file", , vRefNum
Print #1, "This is a test file"
Close #1
```

The program above creates a file whose type is "ttro" and whose creator signature is "txt". The Finder recognizes such a file as a SimpleText "read-only" text file. The file will appear with an appropriate icon, and the Finder will launch SimpleText when the user double-clicks the file's icon.

Note:

`Def Open` only applies to files which are opened with the "O" option. It does not affect the type nor creator signature of files which are opened for input only ("I") or for random access ("R").

See Also:

`Open; Files$`

Def OrSICN statement

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
Def OrSICN(x%, y%, sicnID%)
```

Description:

This statements draws (in the current output window) the small icon found in the “SICN” resource whose ID is *sicnID%*. The icon is drawn with its upper left corner at pixel coordinates (*x%*, *y%*). FB^3 looks first in the current resource file; if no “SICN” resource with the specified ID is found there, then other open resource files are searched. If the “SICN” can’t be found anywhere, the results are unpredictable.

Def OrSICN draws the icon in “transparent” mode: the “black” bits of the icon are plotted in black (or whatever the window’s current foreground color is), but the “white” bits of the icon are not plotted (i.e., they do not alter the window’s contents at those pixel positions).

See Also:

```
Def PlotSICN
```

Def Page

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

Def Page

Description:

This statement displays the printer “Style Dialog.” The exact appearance of the Style Dialog depends on which printer is currently selected, but it typically prompts the user to select portrait or landscape mode, paper type, and other information. Your program should execute the Def Page statement when the user selects a “Page Setup...” menu item.

Note:

After your program executes Def Page, you can use the PrCancel function to determine whether the user clicked the Style Dialog’s “Cancel” button, and you can use the PrHandle function to determine the page setup preferences that the user requested.

See Also:

Clear LPrint; Close LPrint; LPrint; Def LPrint; PrHandle

Def PlotSICN**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def PlotSICN(x%, y%, sicnID%)
```

Description:

This statements draws (in the current output window) the small icon found in the “SICN” resource whose ID is *sicnID%*. The icon is drawn with its upper left corner at pixel coordinates (*x%*, *y%*). FB^3 looks first in the current resource file; if no “SICN” resource with the specified ID is found there, then other open resource files are searched. If the “SICN” can’t be found anywhere, the results are unpredictable.

Def PlotSICN draws the icon in “copy” mode: the “black” bits of the icon are plotted in black (or whatever the window’s current foreground color is), and the “white” bits of the icon are plotted using the window’s current background color & pattern.

See Also:

```
Def OrSICN
```

Def PrintEditField

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Def PrintEditField(efNum& | %)
```

Revision:

February, 2002 (Release 6)

Description:

This routine extracts the contents of an edit field and fits it to the printed page. If the information is longer than a page, it is divided and placed on multiple pages along with page numbers. When this routine is called, you should *not* have already set the printer as the current output via `Route _toPrinter`. `Def PrintEditField` will handle this internally.

When the Appearance Manager is in use, the `efNum&` parameter is a long integer. With standard BASIC, `efNum%` is a short integer.

See Also:

Edit Field

Def RBox

statement

See the Def <just>Box statements.

Def RemoveStr**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def RemoveStr(str#Handle&, item%)
```

Description:

Removes a string from the block referenced by *str#Handle&*. The block must be in the format of an “STR#” resource, and typically *str#Handle&* will be the handle to an “STR#” resource that your program has previously loaded, or the handle to a block that you intend to add as an “STR#” resource. The *item%* parameter indicates the sequential number of the string you want to remove; 1 indicates the first string in the list. If *item%* is greater than the number of strings currently in the list, then `Def RemoveStr` does nothing.

Note:

Never use `Def RemoveStr` on a purgeable resource, unless you first call `FN HNoPurge(str#Handle&)` to (temporarily) make it un purgeable. If you think the resource may have been purged before you had a chance to call `FN HNoPurge`, then you should also call `Call LoadResource(str#Handle&)` to make sure it’s loaded into memory.

If you are using `Def RemoveStr` to update an existing “STR#” resource, then use `Call ChangedResource(str#Handle&)` after using `Def RemoveStr`, to cause your changes to be written to disk when the resource file is closed or updated.

See Also:

Str#; Def ApndStr; Resource Manager chapter in Inside Macintosh: *More Macintosh Toolbox*

Def SetButtonData

statement

✓ <i>Appearance</i>	✗ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Def SetButtonData (bRefNum&, part%, tagType&, size&, dataPtr&)
```

Revision:

February, 2002 (Release 6)

Description:

Controls created for the Appearance Manager have many more features than the time tested push button. They might contain time and date data, a picture, an icon, or just about anything imaginable. The new `Def SetButtonData` statement provides a tool for setting information belonging to one of the new buttons. There are several parameters.

btnID&
part%

The reference number of the button
Apple's designated part index. Part codes are meaningful only within the scope of a single control definition function. For example, the standard tab control uses part codes 1...*n*, where *n* is the number of tabs, even though those numbers collide with part codes used by other control definitions. The list of part codes includes (but is not limited to) the following:

_kControlNoPart	= 0
_kControlLabelPart	= 1
_kControlMenuPart	= 2
_kControlTrianglePart	= 4
_kControlEditTextPart	= 5
_kControlPicturePart	= 6
_kControlIconPart	= 7
_kControlClockPart	= 8
_kControlListBoxPart	= 24
_kControlListBoxDoubleClickPart	= 25
_kControlImageWellPart	= 26
_kControlRadioGroupPart	= 27
_kControlButtonPart	= 10
_kControlCheckBoxPart	= 11
_kControlRadioButtonPart	= 11
_kControlUpButtonPart	= 20
_kControlDownButtonPart	= 21
_kControlPageUpPart	= 22
_kControlPageDownPart	= 23
_kControlIndicatorPart	= 129
_kControlDisabledPart	= 254
_kControlInactivePart	= 255

A button may contain more than one part, so it is often necessary to specify which part is being called for. For instance, you may need to work with the menu, the icon, or the title of a button.

typeTag&

Apple's identifier used to specifically identify the area of the control with which you wish to work. The items include (but are not limited to) the following list:

```

_kControlBevelButtonContentTag      = _"cont"
_kControlBevelButtonGraphicAlignTag= _"gali"
_kControlBevelButtonGraphicOffsetTag= _"goff"
_kControlBevelButtonLastMenuTag     = _"lmnu"
_kControlBevelButtonMenuDelayTag    = _"mdly"
_kControlBevelButtonMenuHandleTag   = _"mhnd"
_kControlBevelButtonMenuValueTag    = _"mval"
_kControlBevelButtonTextAlignTag    = _"tali"
_kControlBevelButtonTextOffsetTag   = _"Toff"
_kControlBevelButtonTextPlaceTag    = _"tplc"
_kControlBevelButtonTransFormTag    = _"tran"
_kControlEditTextKeyFilterTag       = _"fltr"
_kControlEditTextPasswordTag        = _"pass"
_kControlEditTextSelectionTag       = _"sele"
_kControlEditTextStyleTag           = _"font"
_kControlEditTextTEHandleTag        = _"than"
_kControlEditTextTextTag            = _"text"
_kControlFontStyleTag               = _"font"
_kControlGroupBoxFontStyleTag       = _"font"
_kControlGroupBoxMenuHandleTag      = _"mhan"
_kControlGroupBoxTitleRectTag       = _"trec"
_kControlIconAlignmentTag           = _"algn"
_kControlIconTransFormTag           = _"trfm"
_kControlImageWellContentTag        = _"cont"
_kControlImageWellTransFormTag      = _"tran"
_kControlKeyFilterTag               = _"fltr"
_kControlListBoxDoubleClickTag      = _"dblc"
_kControlListBoxFontStyleTag        = _"font"
_kControlListBoxKeyFilterTag        = _"fltr"
_kControlListBoxLDEFTag             = _"ldef"
_kControlListBoxListHandleTag       = _"lhan"
_kControlPopupButtonExtraHeightTag  = _"exht"
_kControlPopupButtonMenuHandleTag   = _"mhan"
_kControlPopupButtonMenuIDTag       = _"mnid"
_kControlPushButtonCancelTag        = _"cncl"
_kControlPushButtonDefaultTag       = _"dflt"
_kControlStaticTextStyleTag         = _"font"
_kControlStaticTextTextHeightTag    = _"thei"
_kControlStaticTextTextTag          = _"text"
_kControlTabContentRectTag          = _"rect"
_kControlTabEnabledFlagTag          = _"enab"
_kControlTabFontStyleTag            = _"font"
_kControlTabInfoTag                 = _"tabi"

```

size&

The size of the data that is being sent. For instance, if you are sending a handle, you might use `SizeOf (Hndl)` or just the number 4 (since a handle variable is 4 bytes long). If you were passing the contents of the handle, you would use `Fn GetHandleSize(h)` where `h` represents a handle. To pass a string you would use `t$(0)` to send the length of the string.

dataPtr&

A pointer to the data that is to be embedded. If you were passing a handle variable, you might use `@h`. To pass the contents of a handle you would use `[h]`. To pass the contents of a string, you would use `@t$(1)`. *passer son contenu, vous utiliserez [h]. Pour passer le contenu d'une chaîne de caractères, vous utiliserez @chaîne\$(1).*

Example:

One example of using `Def SetButtonData` would be in creating an indeterminate progress bar; a progress bar which indicates a process is on going, but its completion time has not yet been determined. In such a case, a standard progress bar control is used, but `Def SetButtonData` is invoked to change the bar's indicator into the indeterminate state.

```

Appearance Button bRef, _activeBtn, 1, 0, 1, , @r, -
    _kControlProgressBarProc

Dim b As Boolean
Dim err As OSErr

b = _True

Def SetButtonData(bRef, _kControlEntireControl, -
    _kControlProgressBarIndeterminateTag, SizeOf(Boolean), @b)

```

In the example above, we use the part code of `_kControlNoPart` and a tag of `_kControlProgressBarIndeterminateTag` to build the call. We are passing a boolean value to the toolbox, so we first dimension a boolean variable (`Dim b As Boolean`), then pass the size of the variable (`SizeOf(Boolean)`), and finally send a pointer to the variable (`@b`).

Def SetButtonFocus

statement

✓ <i>Appearance</i>	✗ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Def SetButtonFocus (bRefNum&)
```

Revision:

February, 2002 (Release 6)

Description:

This procedure sets the target button to be the focus in the current output window. Subsequent key presses will be directed toward that button. Depending on the attributes used when the button was created, this item will usually be encircled by a focus ring.

See Also:

Button, Button function, Appearance Button; Button& function

Def SetButtonStyle

statement✓ *Appearance*✗ *Standard*✗ *Console***Syntax:**

```
Def SetButtonFontStyle (bRefNum&, controlFontRecord)
```

Revision:

February, 2002 (Release 6)

Description:

This procedure sets up all of the font information associated with an Appearance Manager, control-based edit field. You may build such a field using the `Appearance Button` statement with the type of `_kControlEditTextProc`. Text may be placed in this type of button using `Def SetButtonTextString`.

Before modifying text in an Appearance Manager, control-based edit field, you must create a control font style record. This record is defined in the `TLbx Appearance.Incl` as follows:

```
Begin Record ControlFontStyleRec
  Dim flags      As Short
  Dim font       As Short
  Dim size       As Short
  Dim style      As Short
  Dim mode       As Short
  Dim just       As Short
  Dim ForeColor  As RGBColor
  Dim backColor  As RGBColor
End Record
```

You dimension a local copy of the record like this:

```
Dim cfsr As ControlFontStyleRec
```

The first item that must be filled in for the `controlFontStyleRec` is the `flags` entry which will tell the Appearance Manager which parameters are important. Appropriate values for this item are:

```
_kControlUseFontMask
_kControlUseFaceMask
_kControlUseSizeMask
_kControlUseForeColorMask
_kControlUseBackColorMask
_kControlUseModeMask
_kControlUseJustMask
```

Items may be added together by joining the appropriate constants. For instance, if you wanted to change the font and back color of a field, you would use the following:

```
cfsr.flags = _kControlUseFontMask_kControlUseBackColorMask
```

Example:

The following, fully functional program creates an Appearance Manager, control-based edit field and sets the text to Monaco, bold on a red background:

```

Window 1
Appearance Button 1,1,1,0,1,"",_
                    (10,10)-(200,200),_kControlEditTextProc

Dim cfs As ControlFontStyleRec

cfs.flags          =      _kControlUseFontMask + _
                        _kControlUseFaceMask + _
                        _kControlUseBackColorMask
cfs.font           = _monaco
cfs.style          = _boldBit%
cfs.backColor.red  = -1
cfs.backColor.green = 0
cfs.backColor.blue  = 0

Def SetButtonFontStyle( 1, cfs )

Def SetButtonTextString( 1, "Hello" )

Do
    HandleEvents
Until 0

```

See Also:

```

Def SetButtonData, Button function, Appearance Button,
Fn ButtonTextString$

```

Def SetButtonTextSelection

statement

✓ <i>Appearance</i>	✗ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Def SetButtonTextSelection(bRefNum&, selStart, selEnd)
```

Revision:

February, 2002 (Release 6)

Description:

Use this procedure to set the selection range of an Appearance Manager, control-based edit field. (Such buttons are created using the `Appearance Button` statement with a type like `_kControlEditTextProc.`) The minimum value for `selStart` is zero for the position before the first character. The maximum value for `selEnd` is 32767 (or `_maxInt`). To set the selection to the end of the field, use:

```
Def SetButtonTextSelection( bRefNum&, _maxInt, _maxInt)
```


Def SetButtonTextString

statement

✓ <i>Appearance</i>	✗ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Def SetButtonTextString(bRefNum&, pString$)
```

Revision:

February, 2002 (Release 6)

Description:

This Appearance Manager call changes the text string for a control. The two required parameters are the button's reference number and the new text in the form of a Pascal string.

```
Appearance Button 1,1,,, "",␣
(10,10)-(200,30),_kControlEditTextProc
Def SetButtonTextString( 1, "Editable" )

Appearance Button 2,1,,, "",␣
(10,50)-(200,80),_kControlStaticTextProc
Def SetButtonTextString( 2, "Static" )
```

See Also:

```
Def SetButtonData, Button function, Appearance Button,
Fn ButtonTextString$; Def SetButtonFontStyle
```

Def SetDoubleByte

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

Def SetDoubleByte

Revision:

July 26, 2000 (Release 3)

Description:

FB^3 provides the tools for you to handle double-byte characters for use with applications that support Chinese, Japanese, Vietnamese, and Korean scripts. FB^3 provides three ways of handling scripts. The choices are available in the Preferences window under the Vars tab. Only one selection ("Automate handlers for 1 & 2 byte strings") will allow the use of this statement.

Because automated handlers are in effect at all times, you may wish to have your program temporarily ignore them by using `Def SetSingleByte` and reinstate them later with `Def SetDoubleByte`.

See Also:

`Usr GetDoubleByte`, `Def SetSingleByte`, `Usr GetSingleByte`

Def SetSingleByte**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def SetSingleByte
```

Revision:

July 26, 2000 (Release 3)

Description:

FB^3 provides the tools for you to handle double-byte characters for use with applications that support Chinese, Japanese, Vietnamese, and Korean scripts. FB^3 provides three ways of handling scripts. The choices are available in the Preferences window under the Vars tab. Only one selection ("Automate handlers for 1 & 2 byte strings") will allow the use of this statement.

Because automated handlers are in effect at all times, you may wish to have your program temporarily ignore them by using `Def SetSingleByte` and reinstate them later with `Def SetDoubleByte`.

See Also:

```
Def SetDoubleByte; Usr GetDoubleByte, Def GetSingleByte
```

Def SetWindowBackground**statement**✓ *Appearance*✗ *Standard*✗ *Console***Syntax 1:**

```
Def SetWindowBackground(_backgroundConstant, applyNowBool)
```

Syntax 2:

```
Def SetWindowBackground(rgbColor, applyNowBool)
```

Revision:

February, 2002 (Release 6)

Description:

Using Syntax 1, this sets the output window's background to be auto-refreshing. It even maintains the background after the closing of an edit field or button without requiring a window update. This also insures that the proper background is used whether your application is operating in OS 9 or X. This only works with the Appearance Runtime.

Syntax 2 is used to set a background color for the window. The color is a standard RGBcolor record. The *applyNowBool* parameter is non-zero if the window is to be updated immediately and false if the change will take place the next time anything is drawn in the window.

Acceptable values for the *_backgroundconstant* are:

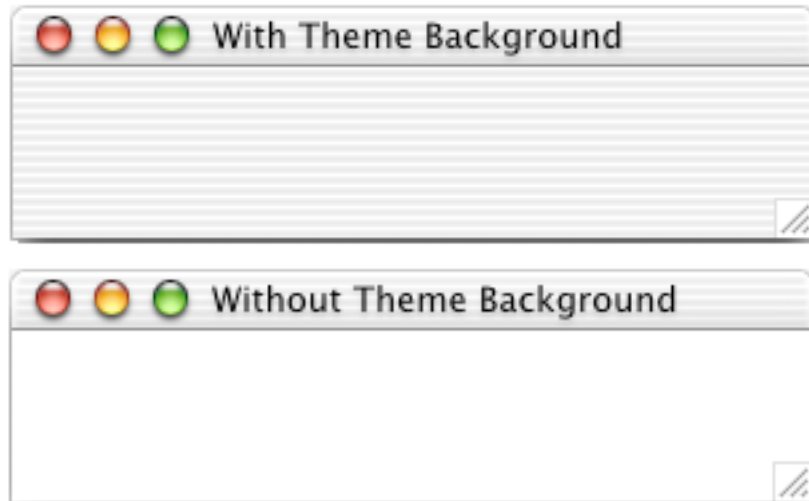
<i>_kThemeActiveDialogBackgroundBrush</i>	(1)
<i>_kThemeInactiveDialogBackgroundBrush</i>	(2)
<i>_kThemeActiveAlertBackgroundBrush</i>	(3)
<i>_kThemeInactiveAlertBackgroundBrush</i>	(4)
<i>_kThemeActiveModelessDialogBackgroundBrush</i>	(5)
<i>_kThemeInactiveModelessDialogBackgroundBrush</i>	(6)
<i>_kThemeActiveUtilityWindowBackgroundBrush</i>	(7)
<i>_kThemeInactiveUtilityWindowBackgroundBrush</i>	(8)
<i>_kThemeListViewSortColumnBackgroundBrush</i>	(9)
<i>_kThemeListViewBackgroundBrush</i>	(10)
<i>_kThemeIconLabelBackgroundBrush</i>	(11)
<i>_kThemeListViewSeparatorBrush</i>	(12)
<i>_kThemeChasingArrowsBrush</i>	(13)
<i>_kThemeDragHiliteBrush</i>	(14)
<i>_kThemeDocumentWindowBackgroundBrush</i>	(15)
<i>_kThemeFinderWindowBackgroundBrush</i>	(16)

Example:

The following code builds a window and sets the background. If the same code were run under system 9, the window with the theme background would be solid gray.

Window 1

```
Def SetWindowBackground(¬  
    _kThemeActiveDialogBackgroundBrush, _zTrue)  
Do  
    HandleEvents  
Until 0
```



Windows with and without theme backgrounds

See Also:

Appearance Window

Def ShadowBox**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def ShadowBox({rect | #rectAddr&})
```

Description:

This statement draws a box with a single-pixel drop shadow, and erases the interior of the box. The box is drawn just outside of the rectangle specified by *rect* (which should be an 8-byte variable such as a *Rect* type) or pointed to by *rectAddr&*. The box can be used as a border for text, pop-up menu items, and other things. The box is not automatically refreshed, unless you're using the Console runtime. Otherwise; your program should explicitly re-draw it (call `Def ShadowBox` again) when you receive an update event for the window.

Console Behavior:

When you use the Console runtime, `Def ShadowBox` switches to the Graphics Window before executing.

Example:

CD Example: `Def ShowdowBox.bas`

See Also:

```
Edit Field; Def <just>Box; Def ShowPop
```

Def ShowPop**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def ShowPop({rect | #rectAddr&})
```

Description:

This statement draws a pop-up menu triangle (▼) in the upper-right corner of the rectangle specified by *rect* (which should be an 8-byte variable such as a `Rect` type) or pointed to by *rectAddr&*. This triangle is most often used with pop-up menus. Unless you're using the Console runtime, the triangle is not automatically refreshed. Your program should explicitly re-draw it (call `Def ShowPop` again) when you receive an update event for the window.

Console Behavior:

When you use the Console runtime, `Def ShowPop` switches to the Graphics Window before executing.

See Also:

```
Def <just>Box; Def ShadowBox
```

Def Tab**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:****Def Tab** [=] *fieldWidth***Description:**

This statement sets the default print field width to *fieldWidth* characters. This affects the number of space characters which are output when a comma is encountered in subsequent `Print`, `Print#` and `LPrint` statements. When an item in such a statement follows a comma, FB³ outputs enough space characters so that each new item begins in a new print field which is *fieldWidth* characters wide. *fieldWidth* must be between 1 and 255. When the program starts, the default print field width is 16 characters.

The value of the print field width affects comma-delimited items whether they're sent to a disk file or to a display device.

See Also:`Width`

Def TitleRect**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def TitleRect(title$, titlePosition%, {rect|#rectAddr&})
```

Description:

This statement draws and titles a rectangle in the current output window, as shown in the figure below. This statement is useful for creating title rectangles around button groupings (especially with radio and checkbox buttons).

The *titlePosition%* parameter specifies the title's horizontal offset (in pixels) from the left side of the rectangle. The rectangle is given by the 8-byte structure contained in the variable *rect* or pointed to by *rectAddr&*.

The rectangle is drawn using the current pen pattern and pen size. The title is drawn using the current font settings. In general, you will want to set the text copy mode to `_srcCopy` in order to erase the part of the rectangle where the text appears.

Unless you're using the Console runtime, the titled rectangle is not automatically refreshed. Your program should explicitly re-draw it (call `Def TitleRect` again) when you receive an update event for the window.

**Example:**

CD Example: `Def TitleRect.bas`

Console Behavior:

When you use the Console runtime, `Def TitleRect` switches to the Graphics Window before executing.

See Also:

`Text`; `Pen`; `Button statement`; `Def <just>Box`

Def Toggle

statement✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def Toggle({var% | #intPtr&})
```

Description:

This statement toggles the value of the short integer contained in the variable *var%* or pointed to by *intPtr&*. It sets the integer to `_false` (0) if it was initially non-zero, and sets it to `_zTrue` (-1) if it was initially zero. This is useful for toggling the value of a variable between `_zTrue` and `_false`. `Def Toggle(var%)` is functionally equivalent to the following code:

```
Long If var% = 0
    var% = -1
Xelse
    var% = 0
End If
```

Def TransitionRect**statement**✓ *Appearance*✗ *Standard*✗ *Console***Syntax:**

```
Def TransitionRect(left%, Top%, right%, botTom%)
```

Revision:

February, 2002 (Release 6)

Description:

This statement is valid only for the Appearance Runtime. It allows specification of a rectangle to be used as the starting point when a window is opened. When this rectangle is supplied, the Appearance Manager animates a rectangle as the window is zoomed open or is hidden. Some special values are appropriate.

To eliminate transition effects, set all parameters to zero.

```
Def TransitionRect( 0, 0, 0, 0 ) // no animation
```

To animate from the center of the current window, set all parameters to -1.

```
Def TransitionRect( -1,-1,-1,-1 ) // animate to/from Window's center
```

To animate from specified set of global coordinates, specify numeric expressions as parameters.

```
Def TransitionRect( 100, 100, 101, 101 )
```

Note:

The transition rectangle is reset to (0,0,0,0) after any window statement that shows or hides a window is executed.

Def Truncate

statement✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Def Truncate ({var$| #stringPtr&})
```

Description:

This statement strips any trailing space characters from the string contained in variable *var\$* (which must be a string variable), or pointed to by *stringPtr&* (which must be a long integer expression or a `Pointer` variable). The resulting string is stored back into *var\$*, or into the buffer pointed to by *stringPtr&*.

Example:

```
tmp$ = "Hello      "
Print "*"; tmp$; "*"
Def Truncate (tmp$)
Print "*"; tmp$; "*"
```

Program output:

```
*Hello      *
*Hello*
```

See Also:

Left\$; Right\$; Mid\$

Def WindowCategory

statement✓ *Appearance*✗ *Standard*✗ *Console***Syntax:**

```
Def WindowCategory (wNum&, category&)
```

Revision:

February, 2002 (Release 6)

Description:

Pre-Appearance versions of the runtime used a window class that was inserted when the window was built. This is a common phenomena to users of Program Generator because of the frequent use of the `gWhichClass` global. In Carbon, the term window class has an entirely different meaning and should not be confused with user assignable FB classes.

For this reason, the term category has been substituted for class and the statement that sets the category has been separated from the window statement.

To set the window category:

```
Def WindowCategory (_myWnd, category)
```

You may use `OSTypes` if that helps your code remain easier to read. For example, an editor window might be set up as the following manner:

```
Def WindowCategory (_myWnd, _"Edit")
```

The category is retrieved with the following code:

```
// retrieve category of active window
category& = Window( _activeWCategory )

// retrieve category of output window
category& = Window( _outputWCategory )
```

This type of categorization makes it easy to create several windows that have unspecific set of behaviors.

Note:

`_activeWCategory` and `_outputWCategory` have the same values as `_activeWClass` and `_outputWClass`

See Also:

Appearance Window; Window function

Def WindowReposition

statement

✓ <i>Appearance</i>	✗ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Def WindowReposition(wRefNum&, parentRefNum&, positionMethod)
```

Revision:

February, 2002 (Release 6)

Description:

Windows are often laid out in relation to other elements on a monitor. For instance, you may want a window to be centered on the main monitor or to be cascaded in relation to a parent window. This procedure provides a mechanism for establishing relative window positions.

- wRefNum&*FB's window reference number
- parentRefNum&*The parent window's reference number. This will be zero for elements that are not related to a parent window.
- positionMethod*One of the following values:

_kWindowCenterOnMainScreen

_kWindowCenterOnParentWindow

_kWindowCenterOnParentWindowScreen

_kWindowCascadeOnMainScreen

_kWindowCascadeOnParentWindow

_kWindowCascadeOnParentWindowScreen

_kWindowAlertPositionOnMainScreen

_kWindowAlertPositionOnParentWindow

_kWindowAlertPositionOnParentWindowScreen

See Also:

```
Appearance Window; Def NewWindowPositionMethod; Def TransitionRect
```

Def<type>**statements**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

DefSng      letter1 [-thruLtr1] [,letter2 [-thruLtr2]...]
DefDb1      letter1 [-thruLtr1] [,letter2 [-thruLtr2]...]
DefStr      letter1 [-thruLtr1] [,letter2 [-thruLtr2]...]
DefInt      letter1 [-thruLtr1] [,letter2 [-thruLtr2]...]
DefLong     letter1 [-thruLtr1] [,letter2 [-thruLtr2]...]

```

Description:

A `Def<type>` statement specifies the data type for subsequently occurring variables, arrays and record fields whose names begin with any of a specified set of letters. The `Def<type>` statement only applies to variables, arrays and record fields whose types are not otherwise explicitly declared—that means you can override the effect of a `Def<type>` statement by putting a type-identifier suffix (like “%” or “&`”) at the end of a name, or by declaring the name in a `Dim` statement using an `As` clause.

Each of the *letter* and *thruLtr* parameters should be a letter of the alphabet. A `Def<type>` statement applies to those simple variables, arrays and record fields whose names begin with *letter1*, *letter2*, etc. Whenever a *thruLtr* parameter is included, the statement also applies to names that start with any letter in the range between *letter* and *thruLtr* (inclusive). (For that reason, a *thruLtr* parameter should always occur later in the alphabet than the *letter* parameter it’s paired with.)

`Def<type>` statements are global in scope (that is, they apply to names declared both inside and outside of local functions). If you use `Def<type>` statements within your program, they should appear near the top of your source code. In particular, if a `Def<type>` statement appears farther down in the source than some variable, array or record field that it applies to, the results could be unpredictable.

`Def<type>` statements are “non-executable,” which implies that they should not appear within any kind of “conditional execution” block, such as `For...Next`, `Long If...End If`, `Do...Until`, etc. (but they may be conditionally included or excluded if you put them inside a `Compile Long If` block).

The following table indicates the default type applied by each of the `Def<type>` statements.

<i>Statement</i>	<i>Default type</i>	<i>Equivalent suffix</i>
<code>DefSng</code>	Simple précision	!
<code>DefDbl</code>	Double précision	#
<code>DefStr</code>	Chaîne	\$
<code>DefInt</code>	Entier	%
<code>DefLong</code>	Entier long	&

Example:

CD Example: `Def<type>.bas`

See Also:

`Compile Long If; DefSng <numbytes>; DefStr Long/Word/Byte; Appendix C:
Data Types and Data Representation`

Def Using

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Def Using [=] intlMoneyFormat&
```

Description:

This statement specifies how the `Using` function should interpret certain characters in its pattern string; this affects the format of the string returned by `Using`. The `intlMoneyFormat&` is specified by four characters packed into a long integer. The characters represent: the decimal point, the thousands separator, the list separator, and the currency symbol. After you execute `Def Using`, subsequent numbers formatted with `Using` are formatted as follows:

- A leading dollar sign in the pattern string is replaced with the currency symbol.
- Commas in the pattern string are replaced with the thousands separator.
- A period in the pattern string is replaced with the decimal point.

Example:

The following selects the U.S. numeric format (this is also the default if no `Def Using` statement has been executed):

```
Def Using = _".,;$"
```

The following selects the international numeric format as set by the system or selected by the user (this is the recommended numeric format to use):

```
Def Using = [[Fn IUGetIntl(0)]]
```

Note:

The international numeric format returned by `Fn IUGetIntl` may include a currency symbol which consists of more than one character. `Def Using` will pick up only the first character of the currency symbol.

See Also:

`Using`

DefStr Long/Word/Byte**statements**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:****DefStr** {**Long**|**Word**|**Byte**}**Description:**

These statements affect the way several integer-to-string functions format their return values. Basically, they affect whether the string functions interpret their arguments as 32-bit, 16-bit or 8-bit integers. The statements are global in scope, and apply to all subsequent calls to the affected string functions, until another `DefStr {Long|Word|Byte}` statement is executed. When the program starts, `DefStr Long` is in effect. The following table shows how the statements affect the return values of the various string functions.

	Bin\$	Hex\$	Oct\$	Uns\$	Mki\$
DefStr Long (Default)	Returns 32 characters.	Returns 8 characters.	Returns 11 characters.	10 characters; adds 2^{32} , if $\text{arg} < 0$	Returns 4 characters.
DefStr Word	Returns 16 characters.	Returns 4 characters.	Returns 6 characters.	5 characters; adds 2^{16} , if $\text{arg} < 0$	Returns 2 characters.
DefStr Byte	Returns 8 characters.	Returns 2 characters.	Returns 3 characters.	3 characters; adds 256, if $\text{arg} < 0$	Returns 1 character.

When `DefStr Byte` is in effect, the string functions may not return the expected result if the integer argument lies outside of the range -255 to $+255$. Likewise, when `DefStr Word` is in effect, the string functions may not return the expected result if the integer argument lies outside of the range -65535 to $+65535$.

See Also:`Bin$; Hex$; Oct$; Uns$; Mki$`

Delay statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Delay count
```

Description:

This statement causes program execution to pause for *count* milliseconds (a millisecond is a thousandth of a second). Delay timing is accurate to the nearest “tick,” which is about 16.7 milliseconds.

Example:

To pause a program for approximately 5 seconds, use the following code:

```
Delay 5000
```

The following built-in constants are useful for producing delays of various durations:

<code>_sec</code>	<code>= 1000</code>	<code>' (1 second)</code>
<code>_secHalf</code>	<code>= 500</code>	<code>' (1/2 second)</code>
<code>_secQuarter</code>	<code>= 250</code>	<code>' (1/4 second)</code>
<code>_secTenth</code>	<code>= 100</code>	<code>' (1/10 second)</code>
<code>_sec60th</code>	<code>= 17</code>	<code>' (about 1 tick)</code>
<code>_secTick</code>	<code>= 17</code>	<code>' (about 1 tick)</code>

Note:

The `Delay` statement ties up the CPU, so that no other processes can execute while the `Delay` is occurring (this limitation does not apply if you’re using the Console runtime). If you intend to implement a very long delay, then consider instead using a timing loop that includes calls to `HandleEvents`.

See Also:

```
Timer; Time$; Date$; Fn TickCount Toolbox function; HandleEvents
```

Dialog**function**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```

evnt%|& = Dialog(0)
id%|& = Dialog(evnt)

```

Revision:

February; 2002 (Release 6)

Description:

The `Dialog` function returns information about the next event (if any) in FutureBASIC's internal `Dialog` queue. An FB^3 `Dialog` event consists of two parts: an "event type" number, which is returned when you call `Dialog(0)`, and "event id" number, which is returned by `Dialog(evnt)` (where *evnt* is the event type number which was returned by `Dialog(0)`). The "event type" identifies what kind of event occurred, and the "event id" provides some secondary information about it.

In the *Appearance* runtime, `Dialog` functions return long integers instead of shorts.

Note that the name "Dialog function" is a bit of a misnomer. It has nothing to do with dialog boxes or dialog controls. FB^3 `Dialog` events consist of a wide range of events, mostly related to the user's interaction with the currently active window.

Normally, you will call `Dialog(0)` and `Dialog(evnt)` from within a dialog-event handling function that your program has designated via the `On Dialog` statement. Your dialog-event handling function should then respond to the event as appropriate.

To make sure that events are properly posted to the `Dialog` queue, your program should call `HandleEvents` periodically. Among other things, `HandleEvents` checks the system event queue; translates any applicable system events into FB^3 `Dialog` events and posts them into the `Dialog` queue; and calls your dialog-event handling function (once per `Dialog` event). `HandleEvents` will also call your dialog-event handling function if your program has posted an event of type `_userDialog` (see the `Dialog` statement).

The following pages list the various types of events returned by the `Dialog` function. In these tables, "Event Type" refers to the number returned by `Dialog(0)`, and "id" refers to the number returned by `Dialog(evnt)`.

Window Events

<i>Event type</i>	<i>Value</i>	<i>Description</i>	<i>ID</i>
<code>_wndClick</code>	3	User clicked in the content area of an inactive window. See notes below.	Window ID of the clicked window
<code>_wndClose</code>	4	User clicked the go away box of an active window. See notes below.	Window ID of the clicked window.
<code>_wndRefresh</code>	5	A window has been resized, or made visible, or a previously obscured part of the window has been uncovered.	Window ID of the window needing a refresh.
<code>_wndActivate</code>	18	A previously inactive window has been made active, or a previously active window has been made inactive. (The active window is the frontmost window and is usually highlighted differently from inactive windows.) See notes below.	If <code>ID</code> is positive, the window with that <code>ID</code> number has been made active. If <code>ID</code> is negative, the window whose <code>ID</code> number is <code>ABS (ID)</code> has been made inactive.
<code>_wndZoomIn</code>	8	User clicked in the zoom box of an active window which is currently in the zoomed out state. See notes below.	Window ID of the clicked window.
<code>_wndZoomOut</code>	9	User clicked in the zoom box of an active window which is currently in the zoomed in state. See notes below.	Window ID of the clicked window.
<code>_wndResized</code>	30	This is an updated version of the older preview event and tells your program that a window was resized. (Appearance Manager only)	Window ID of the resized window.
<code>_inToolBarButton</code>	13	User clicked in the toolbar attached to a window. (Carbon only)	Window ID of the window owning the toolbar.

About `_wndClick` and `_wndActivate` events.

When the user clicks on the content region of an inactive window, FB³ posts a `_wndClick` event, but does not automatically activate the window (your program should execute a `Window` statement in response to the `_wndClick` event, if you want the window to become active).

When the user clicks on the structure region of an inactive window, FB³ posts a `_wndActivate` event, and automatically activates the window the next time your program calls `HandleEvents`.

When you execute a `Window` statement for an inactive window, FB³ posts a `_wndActivate` event, and activates the window immediately.

A `_wndActivate` event can also be generated in response to other actions: for example, when the active window closes and forces another window to become active; or when your application is brought from the background to the foreground.

`_wndActivate` events often occur in pairs: one with a positive “id” value (an inactive window has become active), and one with a negative “id” value (the previously active window has become inactive).

Note: Whenever FB^3 activates a window (for any reason), the newly-active window also becomes the output window (i.e., all new drawing and text commands are sent to that window). If you want to make sure that your program's output doesn't inadvertently get re-directed to the newly activated window, then your program should watch for `_wndActivate` events, and respond by setting the output window (via the `Window Output` statement, or the `SetPort` or `SetGWorld` Toolbox procedures) as appropriate.

Note: Window activation, `_wndActivate` events and `_wndClick` events occur somewhat differently if one or more window has its `_keepInBack` attribute set. See the `Window` statement for more information.

About `_wndClose`, `_wndZoomIn` and `_wndZoomOut` events

When the user clicks on the active window's "close box," FB^3 generates a `_wndClose` event, but does not automatically close the window (your program should execute a `Window Close` statement in response to the `_wndClose` event, if you want the window to close).

When the user clicks on the active window's "zoom box," FB^3 generates a `_wndZoomIn` or `_wndZoomOut` event, and automatically resizes the window the next time your program calls `HandleEvents`.

Disk Insert Events

<i>Event type</i>	<i>Value</i>	<i>Description</i>	<i>ID</i>
<code>_evDiskInsert</code>	17	A floppy disk, tape, or other removable storage medium was inserted into a drive. (Note: some removable-media devices use their own reporting mechanism and won't generate a disk-insert event.	The drive ID number. (Does not function in Appearance Runtime)

Button/ScrollBar Event

<i>Event type</i>	<i>Value</i>	<i>Description</i>	<i>ID</i>
<code>_btnClick</code>	1	User clicked a button or moved a scrollbar thumb.	Button ID number or scrollbar ID number.
<code>_toolBarClick</code>	33	User clicked in the show/hide toolbar button of the window title bar (OS X only)	Window ID number

Contextual Menu Events

<i>Event type</i>	<i>Value</i>	<i>Description</i>	<i>ID</i>
<code>_cntxtMenuClick</code>	24	The user has clicked in a window with the control key modifier pressed. See the <code>Menu</code> statement for additional information.	The window number.

Edit Field Events

Note: these events apply only to editable (non-static) edit fields, and only in the current output window.

<i>Event type</i>	<i>Value</i>	<i>Description</i>	<i>ID</i>
<code>_efClick</code>	2	User clicked in an inactive edit field, and FB^3 activated the field.	Edit field ID of the clicked field.
<code>_efReturn</code>	6	User pressed return while a "noCR" type edit field was active.	Edit field ID of the active field.
<code>_efTab</code>	7	User pressed the tab key while an edit field was active	Edit field ID of the active field.
<code>_efShiftTab</code>	10	User pressed shift-tab while an edit field was active	Edit field ID of the active field.
<code>_efClear</code>	11	User pressed the clear key while an edit field was active.	Edit field ID of the active field.
<code>_efLeftArrow</code>	12	User pressed the left arrow key when the insertion point in the active edit field was already to the left of the first text character.	Edit field ID of the active field.
<code>_efRightArrow</code>	13	User pressed the right arrow key when the insertion point in the active edit field was already to the right of the last text character.	Edit field ID of the active field.
<code>_efUpArrow</code>	14	User pressed the up arrow key when the insertion point in the active edit field was already in the top line of text.	Edit field ID of the active field.
<code>_efDownArrow</code>	15	User pressed the down arrow key when the insertion point in the active edit field was already in the bottom line of text.	Edit field ID of the activated field.
<code>_efSelected</code>	26	A user click or tab key press has changed the focus. (<i>Appearance Manager only</i>)	Edit field ID of the activated field.

Picture Field Event

Note: This event is only reported for picture fields whose `type` parameter equals `_framed`, `_framedNoCR`, `_noFramed`, or `_noFramedNoCR`.

<i>Event type</i>	<i>Value</i>	<i>Description</i>	<i>ID</i>
<code>_efClick</code>	2	User clicked in a picture field.	Picture field ID of the clicked field.
<code>_pfClick</code>	25	User clicked in a picture field. (<i>Appearance Manager only</i>)	Picture field ID of the clicked field.

Key Press Event

Note: This event is not reported if there is an active edit field in the current output window or if you are running under the Appearance Compliant runtime. To capture keypress events in the active edit field, use the `TEKey$` function. To make all edit fields in the current window inactive, use the `Edit Field 0` statement.

<i>Event type</i>	<i>Value</i>	<i>Description</i>	<i>ID</i>
<code>_evCmdKey</code>	24	User pressed a command-key combination which does not match any active menu command key equivalent	ASCII value of the key that was pressed.
<code>_evKey</code>	16	User pressed a key (or key combination) which can be mapped to an ASCII character. In Standard BASIC, this event is only reported when there is no active edit field.	ASCII value of the key (or key combination) that was pressed.

Cursor (Mouse Pointer) Events For the Appearance Runtime

Note: These events might occasionally be missed if the rate at which the cursor moves is too fast compared to the rate at which `HandleEvents` is called.

Note: These events are reported for the output window and for floating windows.

<i>Event type</i>	<i>Value</i>	<i>Description</i>	<i>ID</i>
<code>_cursOverBtn</code>	21	The cursor is over a button. This is the same as the old <code>_cursOver</code> , but is valid only for buttons.	Number of button entered
<code>_cursOverEF</code>	27	The cursor is over an edit field in the active document window or in any floating window.	Number of edit field entered
<code>_cursOverPF</code>	28	The cursor is over a picture field in the active document window or in any floating window.	Number of picture field entered
<code>_cursOverNothing</code>	29	The cursor moved off of a control.	Always zero

Cursor (Mouse Pointer) Events For Standard BASIC

Note: These events might occasionally be missed if the rate at which the cursor moves is too fast compared to the rate at which `HandleEvents` is called.

Note: These events are reported for both active and inactive windows.

<i>Event type</i>	<i>Value</i>	<i>Description</i>	<i>ID</i>
<code>_cursEvent</code>	20	User moved the cursor off of the visible content region of one of the application's windows (either onto the window's structure region, or completely off of the window).	Number of window just entered or zero if not in any FB window.
<code>_cursOver</code>	21	User moved the cursor onto an active button or a non-static edit field, or onto a portion of a window's visible content region other than a button or edit field. (This event can also be generated without actually moving the cursor, by executing the <code>Cursor</code> statement with a negative parameter.)	If <code>ID</code> is zero, the cursor is in the window's visible content region (but not on a button nor an edit field). If <code>ID</code> is positive, the cursor is over an active button or scrollbar which has the given ID number. If <code>ID</code> is negative, the cursor is over a non-static edit field (whether active or inactive) whose ID number is <code>Abs (ID)</code>

Multi-process Events

<i>Event type</i>	<i>Value</i>	<i>Description</i>	<i>ID</i>
<code>_mfEvent</code>	19	An event associated with process-switching has occurred. In System 6, these events were associated with the MultiFinder program; hence the letters "MF".	ASCII value of the key (or key combination) that was pressed. <code>ID</code> is set to one of the following constants: <code>_mfResume</code> (1) <code>_mfSuspend</code> (2) <code>_mfClipboard</code> (3) <code>_msMouse</code> (4) See the notes below for more information.
<code>_FBQuitEvent</code>	31	Your application has received a message to quit. This may be from an Apple Core event or by the user selecting Quit from the application menu.	zero

ID values for “_MFEvent” events

- _mfResume.** Your program has been brought to the front, and the clipboard contents were not altered while your program was in the background.
- _mfSuspend** Your program is being moved to the background or hidden.
- _mfClipboard.** Your program has been brought to the front, and the clipboard contents were altered while your program was in the background. It should be noted that this event cannot be reported under Carbon. Apple suggests the use of the Toolbox function `GetCurrentScrap` for the purpose.
- _mfMouse.** User moved the mouse cursor to an area outside of a special “mouse region” specified by your program. Before this event can be reported, your program must create a region (in global coordinates), and then execute the following statements:

```
Poke Long Event-8, tickFrequency%
Poke Long Event-4, regionHandle&
```

where `regionHandle&` is a handle to the global mouse region, and `tickFrequency%` indicates the desired delay in ticks between successive postings of system events (usually you should set this to 1). The `_mfMouse` event is posted repeatedly for as long as the cursor remains outside of the region. You can later specify a different mouse region by `Poke`'ing a different region's handle into `Event-4`, or you can disable the event as follows:

```
Poke Long Event-4, _nil
```

For as long as a region remains the “active” mouse region, you must not dispose of that region's handle.

Preview Events

<i>Event type</i>	<i>Value</i>	<i>Description</i>	<i>ID</i>
<code>_preview</code>	22	Usually indicates that some other related event is about to be posted.	<p>"ID" is set to one of the following constants:</p> <pre> <pre>_premenuclick (1) _prewndgrow (2) _wndmoved (3) _wnd-sized (4) _efchanged (5) _preefclick (6) _prewndzoomin (7) _prewndzoomout (8) _wnddocwillmove (9)</pre> <p>See notes below for more information.</p> </pre>

ID values for “_preview” events

<code>_preMenuClick</code>	User clicked on a menu in the menubar. This event is reported before the menu actually opens.
<code>_preWndGrow</code>	User clicked on the active window’s size box. This event is reported as soon as the mouse is pressed down.
<code>_wndMoved</code>	User clicked on the active window’s title bar (or its frame, in MacOS 8.x). This event is reported after the mouse button is released.
<code>_wndSized</code>	User resized the window. This event is reported after the size change. See <code>_wndResized</code> for a new dialog event invoked by the Appearance Manager runtime.
<code>_efChanged</code>	User selected the “Clear”, “Cut” or “Paste” option from the Edit menu. This applies only when the Edit menu was created by the <code>Edit Menu</code> statement. The event is reported before the text actually changes.
<code>_preEfClick</code>	User clicked in a non-static edit field in the active window. This event is reported whether the clicked field is the active field or not. The event is returned after the mouse button is released, but before the <code>_efClick</code> event (if any) is reported.
<code>_preWndZoomIn</code>	User clicked in the active window’s Zoom box (while the window was in the “zoomed-out” state). This event is reported after the mouse button is released, but before the window actually changes size (and before the <code>_wndZoomIn</code> event is reported).
<code>_preWndZoomOut</code>	User clicked in the active window’s Zoom box (while the window was the “zoomed-in” state). This event is reported after the mouse button is released, but before the window actually changes size (and before the <code>_wndZoomOut</code> event is reported).

User Dialog Events

<i>Event type</i>	<i>Value</i>	<i>Description</i>	<i>ID</i>
<code>_userDialog</code>	23	The program posted a custom event using the <code>Dialog = expr%</code> statement.	The value of <code>expr%</code> that was specified when the event was posted.

Note:

You can use the `Event%` function and the `Event&` function to retrieve more details about an event returned by the `Dialog` function.

See Also:

`HandleEvents`; `On Dialog Fn`; `Event%` function; `Event&` function

Dialog**statement**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
Dialog = expr%
```

Description:

This statement posts a `Dialog` event of type `_userDialog` to FutureBASIC's internal `Dialog` queue. This allows your program to post “custom events” to itself. After you post a `_userDialog` event, you can use the `Dialog` function to subsequently read the event from the queue (normally you'll do this from within the dialog-event handling function designated by an `On Dialog` statement). If you retrieve the event from the queue as follows:

```
evnt = Dialog(0)
id = Dialog(evnt)
```

then `evnt` will be set to the value `_userDialog` (which equals 23), and `id` will be set to the value of *expr*% that you specified in the `Dialog` statement.

To learn how to pass additional information associated with an event of type `_userDialog`, see the descriptions of the `Event%` and `Event&` statements, and the `Event%` and `Event&` functions.

See Also:

`Dialog` function; `On Dialog`; `HandleEvents`; `Event%` statement/function;
`Event&` statement/function

Dim statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Dim declaration1 [,declaration2...]
```

Revision:

July 26, 2000 (Release 3)

Description:

(Note: This description applies to `Dim` statements that occur inside of “true record” definition blocks (`Begin Record...End Record`), as well as those which occur outside of record definition blocks. `Dim` statements which occur inside of “pseudo-record” definition blocks (`Dim Record...Dim End Record`) are used somewhat differently; see the `Dim Record` statement for more information.)

Effective with Release 3, a new syntax may be used with `Dim`.

```
Dim As Long a,b,c
Dim As Rect myRect
```

The `Dim As` syntax indicates that all variables that follow are of the same type and size. This is different from `Dim a As Long` which provides variable specification for only one dimensioned variable in the statement.

`Dim` is a non-executable statement that allows the compiler to determine how much storage space should be allocated for the declared variables, arrays and record fields, and identifies their data types. `Dim` can also be used to affect the relative storage locations of the declared variables, arrays and fields.

A declaration can have any of the following forms:

Simple variables:

```
{[@]varName | [maxLen] stringVar$ }[;statExpr]
untypedVar As predefinedType [;statExpr]
untypedVar As userType [;statExpr]
```

Records:

```
untypedVar.constant2 [;statExpr]
untypedVar As recordType [;statExpr]
```

Arrays:

```
{varName | [maxLen] stringVar$}(maxSub1[,maxSub2...]) [;0]
untypedVar (maxSub1 [,maxSub2...]) As predefinedType [;0]
untypedVar (maxSub1 [,maxSub2...]) As userType
untypedVar.constant2 (maxSub1 [,maxSub2...])
```

Pointers:

```
[@] untypedVar As {Pointer To|^|@|.}{predefinedType|recordType}
```

Handles

```
[@] untypedVar As {Handle To|^|^|@@|..}{predefinedType|recordType}
```

Memory alignment declarations:

```
{%|&|&&|&&&|.constant}
```

- `maxLen`, `maxSub1`, `maxSub2` and `statExpr` are (non-negative) static integer expressions.
- `constant` is a (non-negative) literal integer or a symbolic constant; but if a symbolic constant is used, its initial underscore character is omitted.
- `stringVar$` is a variable name that ends with a “\$” type-identifier suffix.
- `varName` is a variable name that may optionally end with a type-identifier suffix.
- `untypedVar` is a variable name that does not end with a type-identifier suffix.
- `userType` is a type name defined in a previous `Begin Record` statement or `#Define` statement.
- `recordType` is a type name defined in a previous `Begin Record` statement.
- `predefinedType` is one of the following: `Char`, `[Unsigned] Byte`, `[Unsigned] Word`, `[Unsigned] Short`, `[Unsigned] Int`, `UInt16`, `[Unsigned] Long`, `UInt32`, `Point`, `Fixed`, `Rect`, `Handle`, `RGNHandle`, `Str255`, `Str63`, `Str31`, `Str15`, `Double`, `Single`.

If a `Dim` statement appears within a `Begin Globals...End Globals` block, then the scope of the declared variables and arrays is global. If it appears within the scope of a `Local` function or an `EnterProc` procedure block (but not within a `Begin Globals...End Globals` block), then the scope of the declared variables and arrays is local to that function or procedure block. If `Dim` appears outside of all local functions and procedure blocks (and outside of any `Begin Globals...End Globals` block), then the scope of the declared variables and arrays is local to “main.”

Your program can use as many `Dim` statements as you like. The following statement:

```
Dim a, b&, c$, d#
```

is equivalent to this pair of statements:

```
Dim a, b&
Dim c$, d#
```

Certain structures must always be declared in a `Dim` statement, which must appear somewhere above the first line where the structure is used. These structures include:

- Arrays (unless declared in an `Xref` or `Xref@` statement)
- Record variables
- Variables of user-defined types
- `Pointer` variables and `Handle` variables

If the `_dimmedVarsOnly` compiler option is used, then this restriction applies to all variables, except variables which appear in the parameter list of a `Local Fn` statement or `EnterProc` statement.

If the `_dimmedVarsOnly` compiler option is not used, then simple string or numeric variables don't need to be declared in a `Dim` statement. In that case, a simple variable is implicitly declared in the first statement (within its scope) in which the variable appears, and the compiler uses that implicit declaration to assign storage space for the variable.

Storage space for FutureBASIC's built-in types is allocated as follows:

<i>Type</i>	<i>Storage</i>
Entiers sur octet (` , ``); Byte; Char	1 byte
Entiers courts (+,%`); Short; Int; Word	2 bytes
Entiers longs (&,&`); Long	4 bytes
Point; Fixed	4 bytes
Simple précision (!); Single	4 bytes
Double précision (#); Double	(see note)
Rect	8 bytes
Str255	256 bytes
Pointer	4 bytes
Handle	4 bytes

(Note: the storage space for double-precision variables depends on which kind of CPU your program is compiled for: 10 bytes in 68K, 8 bytes in PPC. Use the `SizeOf` function to make a definite determination of these variables' sizes.)

The storage space allocated for a string variable depends on the value of the `maxLen` parameter (which cannot exceed 255). If `maxLen` is omitted, then the most recent `maxLen` specified in the same `Dim` statement is used. If there is no previous `maxLen` specified in the current `Dim` statement, then the value specified by the most recent `Def Len` statement is used. If there is no `Def Len` statement preceding the `Dim` statement, then `maxLen` defaults to 255. String variables declared using the `As Str255` clause always have a `maxLen` value of 255.

Once `maxLen` has been determined for a given string variable, the actual number of bytes allocated for the variable is:

- `maxLen + 1` bytes, if `maxLen` is odd;
- `maxLen + 2` bytes, if `maxLen` is even;

Your program should not assign a string longer than `maxLen` characters to a string variable. The storage space for a record variable equals the sum of the lengths of the record's fields, or the value of `constant2` (in bytes).

The storage space for an array is calculated as follows: If `elSize` is the size in bytes of a single array element, then the space allocated for the entire array is given by the following expression:

```
array size = elSize * (maxSub1 + 1) * (maxSub2 + 1) * ...
```

All the elements in an array are stored in contiguous locations in memory. If the array is multi-dimensional, then the rightmost dimensions change most rapidly as you step through the elements' locations in memory. For example, if you declare an array as follows:

```
Dim p%(3, 2)
```

Then the elements of `p%()` are stored in this order in memory:

```
p%(0,0)
p%(0,1)
p%(0,2)
p%(1,0)
p%(1,1)
p%(1,2)
p%(2,0)
p%(2,1)
p%(2,2)
p%(3,0)
p%(3,1)
p%(3,2)
```

Using the “@” symbol to force RAM storage

To increase processing speed, FB^3 may store some variables in CPU registers rather than in (slower) addressable RAM memory. If you use the `Register On` statement, or the “Use register-based variables” compiler preference is set, FB^3 stores as many long-integer, short-integer and byte variables as possible into CPU registers. The compiler starts with the first variables it encounters in the current scope, and assigns them to registers until no more registers are available.

Sometimes this method is not appropriate. In cases where your program needs to obtain a variable's address, the variable must be stored in addressable RAM memory. When you put an “@” symbol in front of a variable's name in a `Dim` statement, you force all the remaining variables in the `Dim` statement (to the right of the “@”) to be stored in RAM memory rather than in CPU registers. See the `Register On/Off` statements for more information.

Using Memory-Alignment Declarators to adjust storage location

A memory-alignment declarator (`%`, `&`, `&&`, `&&&`) affects the storage location of the next simple variable or record variable which is declared after it, forcing the variable to begin at a byte location whose address is divisible by 2, 4, 8 or 16. This can improve the efficiency of certain low-level operations that involve the variable, and may thereby improve your program's speed. The `.constant2` memory declarator forces an arbitrary number of empty bytes to be inserted between variables.

<code>Dim %, var</code>	Forces <code>vars</code> to begin on a word boundary (address divisible by 2).
<code>Dim &, var</code>	Forces <code>var</code> to begin on a longword boundary (address divisible by 4).
<code>Dim &&, var</code>	Forces <code>var</code> to begin on a double-longword boundary (address divisible by 8).
<code>Dim &&&, var</code>	Forces <code>var</code> to begin on an extended-word boundary (address divisible by 16).
<code>Dim .constant2, var</code>	Forces <code>var</code> to begin at <i>constant</i> bytes

Note: using the above alignment declarators forces all subsequent variables in that `Dim` statement to be stored in addressable memory rather than in CPU registers. See the `Register On/Off` statements for more information.

Using the semicolon (;) syntax to adjust storage location (for non-array variables)

Simple variables and record variables are normally stored in contiguous locations in memory, according to the order in which they're declared. For example, consider the following `Dim` statement:

```
Dim x%, myRect.8, 15 s1$, y&
```

Assuming that none of these variables was implicitly declared above this `Dim` statement, and that they're all located in addressable RAM, then they will appear contiguously in memory, as shown in this diagram:



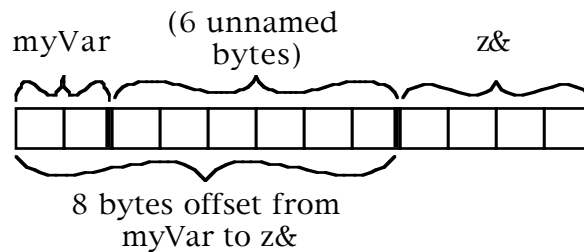
It's sometimes useful to be able to make an adjustment to this storage scheme. This can be done by using the semicolon syntax. When a declaration of the form `var; constant1` appears in a `Dim` statement, the compiler is instructed to apply an offset of `constant1` bytes between the beginning of `var` and the beginning of the next declared simple variable or record variable. This can be useful in a couple of different ways:

- If `constant1` is larger than the storage size of `var`, then extra (unnamed) bytes are reserved after `var`. For example:

```
Dim myVar;8, z&
```

FUTUREBASIC REFERENCE

This instructs the compiler to allocate storage as follows (assuming that untyped variable names like `myVar` default to 2-byte short integers):

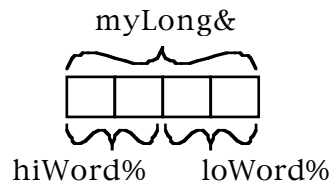


Here, even though `myVar` is actually a 2-byte integer, you can store up to 8 bytes at its location without interfering with other program variables. This is useful in the case of certain older FutureBASIC statements, which require the name of a short integer variable, but which expect a larger data structure to be stored at that variable's address.

- If `constant1` is smaller than the storage size of `var`, then you can force variables to “overlap” in memory. This produces an effect similar to the “union” keyword in C, or the “EQUIVALENCE” statement in Fortran. For example:

```
Dim myLong&;0, hiWord%, loWord%
```

Here the compiler is instructed to put no offset between the beginning of `myLong&` and the beginning of `hiWord%`. The result looks like this:

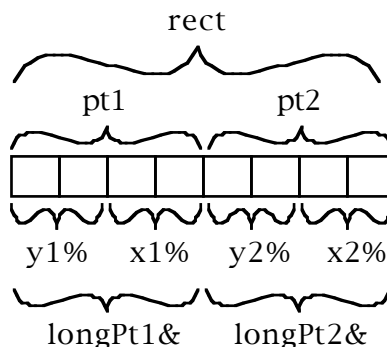


In this example, whenever the program assigns a new value to `myLong&`, the values of `hiWord%` and `loWord%` are correspondingly altered. Likewise, if the program assigns a new value to `hiWord%` or `loWord%`, then the value of `myLong&` is altered.

The following example carries this concept a bit further:

```
Dim rect.8;0, pt1.4;0, longPt1&;0, y1%, x1%
Dim pt2.4;0, longPt2&;0, y2%, x2%
```

Here the variables are stored like this:



In this example, these 8 bytes of storage are known by a number of different names! If we think of the 8 bytes as representing a standard QuickDraw rectangle structure, then: `rect` represents the full structure; `pt1` and `pt2` represent the points at the upper-left and lower-right of the rectangle; `longPt1&` and `longPt2&` are long-integer synonyms for `pt1` and `pt2`; and `x1%`, `y1%`, `x2%` and `y2%` represent the individual coordinates of the two corner points.

If you use the semicolon syntax in combination with a memory-alignment declarator, then their effects are combined. For example, if you declare the following:

```
Dim 13 z$;7, &, x%
```

then the variable `x%` begins at the first longword boundary that is at least 7 bytes past the beginning of variable `z$`.

Note: using the semicolon in a (non-array) declaration forces the variable preceding the semicolon, and all variables following it in the same `Dim` statement, to be stored in addressable memory, rather than in CPU registers. See the `Register On/Off` statements for more information.

Using the “;0” syntax with arrays

The semicolon syntax can be used in a limited way with arrays of simple (non-record) variables. For example, if you declare the following:

```
Dim abc&(10,15);0
```

then the compiler is instructed to align the beginning of the next declared array of simple variables with the beginning of array `abc&()`. That other array need not be of the same simple type, nor have the same dimensions, as `abc&()`. Note that the semicolon syntax is not available for arrays of records. Also, when using the semicolon with arrays, you must follow the semicolon with a zero; you can't use an arbitrary constant in this case.

FB³ stores arrays and non-array variables in two separate sections of memory. You cannot use a semicolon after a non-array variable to adjust the storage location of an array, nor vice-versa.

See Also:

```
Begin Globals...End Globals; Dim Record; Begin Record...End Record;  
Register On/Off
```

Dim Dynamic

See the `Dynamic` statement

statement

Dim End Record

See the `Dim Record` statement

statement

Dim Record

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

Dim Record recordName
    Dim declaration1 [,declaration2...]
    [Dim declaration3 [,declaration4...]]
    :
Dim End Record recIdentifier

```

A *declaration* can have any of the following forms:

```

identifier[typeSuffix] [;statExpr]
[maxLen] identifier$ [;statExpr]
identifier As {predefinedType|userType} [;statExpr]
identifier.constant [;statExpr]
{% | & | && | &&&}

```

- *identifier* is any standard FB³ identifier that begins with a letter and contains only letters and digits.
- *typeSuffix* is any type-identifier suffix.
- *statExpr* is a non-negative static integer expression.
- *maxLen* is a literal or symbolic integer constant in the range 1 through 255.
- *predefinedType* is one of the following: Char, [Unsigned] Byte, [Unsigned] Word, [Unsigned] Short, [Unsigned] Int, UInt16, [Unsigned] Long, UInt32, Point, Fixed, Rect, Handle, RGNHandle, Str255, Str63, Str31, Str15, Double, Single
- *userType* is a type name defined in a previous `Begin Record` statement, or in a previous `#Define` statement.
- *constant* is a non-negative literal integer or a symbolic constant; but if a symbolic constant is used, its initial underscore character is omitted.
- *recIdentifier* is an identifier that begins either with an underscore character or a period.

Description:

The `Dim Record` statement indicates the beginning of a “pseudo-record” definition block (as contrasted with a “true record” definition, which is declared using a `Begin Record...End Record` block). As shown in the syntax description, the block must contain `Dim` statements, and must end with a `Dim End Record` statement. The “pseudo-record” definition block instructs the compiler to generate a new symbolic integer constant from each identifier, and from *recIdentifier*. For example, the following block:

```

Dim Record myRecord
    Dim firstField%
    Dim 5 secondField$
    Dim thirdField&
    Dim fourthField.10
Dim End Record .myRecSize

```

is exactly equivalent to these statements:

```

_firstField = 0
_secondField = 2
_thirdField = 8
_fourthField = 12
_myRecSize = 22

```

For this reason, each *identifier*, and *recIdentifier*, must have names which are different from any symbolic constants defined anywhere else in the program, and different from the names of any FB³ predefined symbolic constants. This also implies that none of the *identifier* and *recIdentifier* names in one **Dim Record** block can be the same as any *identifier* or *recIdentifier* names in any other **Dim Record** block.

After these symbolic constants have been generated, you can use the *recIdentifier* constant to declare a “pseudo-record” variable. Using the above example, you could specify the following anywhere after the **Dim Record** block:

```

Dim aRecord.myRecSize

```

This reserves 22 bytes for the variable “aRecord,” and is exactly equivalent to this declaration:

```

Dim aRecord.22

```

You can then think of this 22-byte area as being divided into four “fields”:

- A 2-byte short integer (firstField%)
- A 6-byte (max 5 characters) string (secondField\$)
- A 4-byte long integer (thirdField&)
- A 10-byte sub-record (fourthField)

Notice that the values of the first four symbolic constants generated in the **Dim Record** block represent the byte offset values to the beginnings of these fields. This means that you can use those constants along with FutureBASIC’s “embedded-dot” syntax to access the individual fields within the pseudo-record (see Appendix B: *Variables*, to learn more about the embedded-dot syntax). For example:

```

Print aRecord.thirdField&

```

This instructs FB³ to print the long integer which is located 8 bytes past the beginning of aRecord. Note that you could also have written this statement as:

```

Print aRecord.8&

```

However, using the symbolic constant names generally makes the program easier to read and maintain.

Using the alignment declarators (% , & , && , &&&)

These are used in a way similar to their use in the “standalone” `Dim` statement. An alignment declarator causes the symbolic constant generated from the next declared `identifier` or `recIdentifier` to have a value which is divisible by 2, 4, 8 or 16. For example:

```
Dim Record myRec
  Dim abc.9
  Dim n%
  Dim &, xyz%
Dim End Record .recSize
```

The “&” declarator forces the constant `_xyz` to have a value which is divisible by 4. The constants are generated with these values:

```
_abc      = 0
_n        = 9
_xyz      = 12
_recSize  = 14
```

Using the semicolon (;) syntax

This is used in a way similar to its use in the “standalone” `Dim` statement. A declaration of the form `Dim var1; constant, var2` causes `var1` and `var2` to generate constants which differ by the value of `constant`. For example:

```
Dim Record myRec
  Dim 5 theName$
  Dim theRect;0
  Dim pt1.4
  Dim pt2.4
Dim End Record .recSize
```

This block causes the following constants to be generated:

```
_theName = 0
_theRect = 6
_pt1     = 6
_pt2     = 10
_recSize = 14
```

Using an underscore with recIdentifier

If *recIdentifier* begins with an underscore character rather than with a period, then in addition to generating new symbolic constants, FB^3 also reserves storage space for a single pseudo-record variable (whose name is specified by the *recordName* parameter), as well as a variable for each of the “fields.” Furthermore, the field variables are assigned storage which lies inside the record variable. For example, this block:

```
Dim Record myRecord
  Dim firstField%
  Dim 5 secondField$
  Dim thirdField&
  Dim fourthField.10
Dim End Record _myRecSize
```

is exactly equivalent to these statements:

```
_firstField = 0
_secondField = 2
_thirdField = 8
_fourthField = 12
_myRecSize = 22
Dim myRecord.myRecSize;0
Dim firstField%, 5 secondField$
Dim thirdField&, fourthField.10
```

In this case, while the scope of the symbolic constants is global, the scope of the declared variables depends on where in the program the `Dim Record` block appears.

See Also:

`Dim; Begin Record...End Record`

Dim System

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Dim System staticExpr& [, preferredSize& [, version& ↵
                        [, release& [, aboutStringWith4CharPrefix$]]]]
```

Revision:

August, 2002 (Release 7)

Description:

This is a non-executable statement which you can use to increase the amount of memory that the compiler assigns to your compiled application. The *staticExpr* parameter must be a positive static integer expression; it indicates an amount of memory in bytes.

FB³ attempts to calculate an optimum amount of memory to assign to your application. If you find that this default amount is not enough (for example, if your program dynamically allocates large blocks of memory), use `Dim System` to increase the amount. `Dim System` increases the “Suggested Size”, “Minimum Size” and “Preferred Size” values that appear in the application’s “Get Info” window in the Finder; however, the amounts by which those values increase will in general not be exactly equal to *staticExpr*. If you find that your program is still running out of memory, increase the value of *staticExpr*.

A program may contain several `Dim System` calls and each will add to the amount of memory set aside for the application. Individual includes may each request extra memory for known requirements like offscreen GWorlds or large data handles.

Minimum/Preferred Sizes

The values used for `Dim System` are added together to help establish a minimum size for the application's partition. At the same time, FB³ establishes a preferred size that is 1.5 times as large as the minimum size when no *preferredSize&* parameter is specified.

Parameters:

<i>staticExpr&</i>	The number of extra bytes to be allocated in addition to those calculated as necessary by the compiler.
<i>preferredSize&</i>	The amount of memory that your application would prefer to have if sufficient RAM is available.

The next parameters require that an existing vers resource (ID 1) be included in your project (See Resources statement and/or "Project Manager & Debugger" manual).

<i>version&</i>	Usually specified in hexadecimal format, this version number will appear in the Get Info dialog in the version field provided you have used the <i>aboutStringWith4CharPrefix\$</i> parameter in your Dim System statement. For instance, 0x0123 would indicate version 1.23.								
<i>release&</i>	<p>Usually specified in hexadecimal format, this release number indicates the stage in the creation process of your software. The beginning digit specifies the type of built for the resulting application.</p> <table> <tr> <td>0x8xxx</td><td>Release</td></tr> <tr> <td>0x6xxx</td><td>Beta</td></tr> <tr> <td>0x4xxx</td><td>Alpha</td></tr> <tr> <td>0x2xxx</td><td>Development</td></tr> </table> <p>Additional digits tell which iteration this belongs to. For instance, the third beta release would be represented by 0x6003.</p>	0x8xxx	Release	0x6xxx	Beta	0x4xxx	Alpha	0x2xxx	Development
0x8xxx	Release								
0x6xxx	Beta								
0x4xxx	Alpha								
0x2xxx	Development								
<i>aboutStringWith4CharPrefix\$</i>	this is actually two parameters in one. The first 4 characters of the string are used as a file type. The remaining characters contain information about the program that will appear in the Finder's Get Info dialog.								

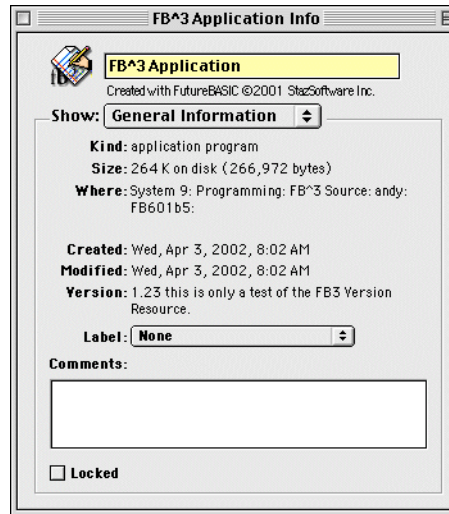
Note:

FutureBASIC creates a vers resource ID 2 when building any application. This vers resource will still be present in the final application even if you use the additional parameters of the Dim System statement to alter an existing vers resource of ID 1.

Example:

```
Dim System 666*1024,777*1024
Dim System ,,0x0123,0x6063,-
        "x.xx this is only a test of the FB3 Version Resource."
```

After using these statements, the finder's Get Info box would look like this:

**Note:**

To set your application's memory size precisely, create a "SIZE" resource with ID = -1, set its "Size" and "Min Size" fields to the values you want, and use the `Resources` statement to include the resource in your program. If you use `Dim System` in this case; `Dim System` will override the values you specified in the "SIZE" resource only if the value it calculates is larger than the value held in the resource.

If you want to make an application that is smaller than FB's suggested minimum size, you may use a negative number to indicate the amount of memory that will be subtracted from the calculated total.

See Also:

`Resources`

Do statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

Do
    [statementBlock]
Until expr

```

Description:

The `Do` statement marks the beginning of a “do-loop,” which must end with an `Until` statement. *statementBlock* represents a block of zero or more executable statements, possibly including other do-loops. When a `Do` statement is encountered, FB³ executes the statements (if any) in *statementBlock*, and then evaluates *expr*. If *expr* is zero, then the process is repeated. The statements in *statementBlock* are repeatedly executed until *expr* evaluates to a nonzero value, at which point the loop exits and the next statement after `Until` is executed. Typically, *expr* is an expression involving logical operators, which is evaluated either as `_zTrue` (−1) or `_false` (0). See the `If` statement for more information about *expr*.

Note that the statements in *statementBlock* are always executed at least once. If you want to use a looping structure which may possibly skip over the *statementBlock* without executing it, consider using a `While/Wend` loop.

See Also:

`For...Next`; `While...Wend`; `If`

Dynamic statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
[Dim] Dynamic arrayName(maxSub1[,maxSub2 ...]) [As dataType]
```

Revision:

July, 2002 (Release7)

Description:

The `Dynamic` syntax is an alternate version of `Dim` that allows for arrays that can grow as needed. The constant expression used in the parenthesis should be the theoretical maximum that will ever be needed by the array. Since it is not possible for a dynamic array to have a value that is out of range, this value is used only to prevent the runtime from reporting out of range errors. Dynamic arrays may only be created and used as global arrays. Do not attempt to dimension them inside of a `Local Fn.`

Example:

```
// 1000 elements max
Dynamic myIntArray(1000)

// 2 gig elements max
Dynamic hugeEmployeeRecAry(_maxLong) As employeeRec

// 32,767 elements max
Dynamic arrayOfRects(_maxInt) As Rect
```

The `maxSub1`, `maxSub2` etc. values must be positive static integer expressions. However, since `Dynamic` does not actually allocate any memory, the declared subscripts are used somewhat differently than in a `Dim` statement. The second and subsequent subscripts (if any) determine the internal structure of the array, and space for them will be fully allocated for each element dynamically referenced in the first subscript. But the value of the first subscript (`maxSub1`) is ignored, and may be arbitrarily set to any value greater than zero. You can actually reference array elements greater than `maxSub1`, so long as adequate RAM is available to allocate the memory required.

Auto Grow

When a dynamic array is dimensioned, the only thing set aside in memory is an eight byte record that tracks items used and space allocated. As soon as you insert information into the array, it begins to grow. You can set things up so that the array grows one element at a time. This has a slight advantage of saving a few bytes, but an overall disadvantage of taking longer to execute as the grow routines must be called much more often.

You can control this value by setting a global variable: `gFBDynamicGrowInc&`. The default value for this global variable is 10. If the value is left at 10, then the following actions would take place as information was placed in array elements:

```
// 1000 elements max
Dynamic myIntArray(1000)
// Current handle size is zero

myIntArray(1) = 0
// Current size has jumped to 10 elements
// (0 through 9 at 2 bytes per element)

myIntArray(8) = 23
// Handle size does not change.
// There is already sufficient room
```

You may force the handle to grow to a maximum expected size. This keeps the runtime from invoking several operations to increase the size of the handle. It does not prevent the runtime from making additional checks to see that future additions fit in the allocated space and it does not prevent further resizing. When the size of a dynamic array is expanded, the newly created elements are set to zero.

You may determine how many elements are being used in a dynamic array using the following method:

```
Dynamic 15 gTest$(100)

address& = @gTest
nextindex& = address&.AuToXREFCurr&
```

Note:

Dynamic arrays may only be global in nature and may not be dimensioned inside of a `Local Fn.`

Warning:

Dynamic arrays may not be used to dimension an array of handles.

See Also:

`Compress Dynamic; Kill Dynamic; Read Dynamic; Write Dynamic`

Edit

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

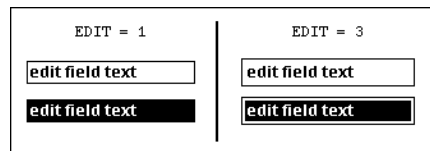
```
Edit = fieldMargin
```

Description:

Use this statement to specify the margin (in pixels) between the “view rectangle” and the frame, in all framed edit fields. (The “view rectangle” is the text-bounding rectangle you specify in the `Edit Field` statement). The specified *fieldMargin* applies to all subsequently created framed fields in all windows. It also applies to all existing framed fields; however, the frames of existing fields are not redrawn with the new margin immediately. To force existing visible frames to be redrawn after you execute the `Edit` statement, use the Toolbox procedure `InvalRect` to identify the fields’ rectangles as requiring an update: `FB^3` will then redraw them the next time your program executes a `HandleEvents` statement.

Before the first `Edit` statement is executed, the default *fieldMargin* value is 1 pixel.

Example:



See Also:

```
Edit Field; SetSelect
```

Edit\$**function**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```

fieldContents$|container$$ = Edit$(fieldID)
fieldContents$|container$$ = Edit$(fieldID, lineNumber)
fieldContents$|container$$ = Edit$(fieldID, -1)
fieldContents$|container$$ = Edit$(fieldID, selStart, selEnd)

```

Revision:

May 30, 2000 (Release 3)

Description:

This function returns the contents of the specified edit field in the current output window. If the edit field contains more than 255 characters of text and a string is specified to receive the information, then only the first 255 characters are returned. Where a container is the specified target, there is more than enough room to hold the contents of a field. Use the `Get Field` statement if you need to retrieve style information along with the text of a styled edit field.

If a single parameter is used, the `Edit$` function attempts to return as much of the entire edit field as will fit into the target variable.

If a second parameter of -1 is added, the function returns the current selection in whole or part depending on the size of the target variable. A second parameter that consists of a numeric value specifies the line number that is to be returned.

Where three parameters are used, the second and third values are used to specify the starting and ending points of text to be captured. If the target variable is a string instead of a container, no more than 255 bytes of information can be returned.

In all cases, FB³ ensures that variables are not overflowed. This is important because we are working with three different sizes of items here.

Item	Size
Pascal String	255 bytes + length Byte
Edit Field	32,767 bytes
Container	2 gigabytes

If *fieldID* refers to a picture field, then the `Edit$` function returns the `pictID$` string that was specified in the `Picture Field` statement. You can use this to identify the handle or resource that contains the picture.

If there is no edit field nor picture field with an ID of *fieldID* in the current output window, the `Edit$` function returns a null string.

See Also:

`Get Field`; `Edit Field`; `Edit$ statement`; `Window(_efNum)`; `Picture Field`

Edit\$**statement**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```

Edit$(efID) = ¬
    string$|&ZTXTHandle&|%TEXTresID%|#container$$¬
    font, size, style, mode, red, green, blue
Edit$(efID, lineNumber) = ¬
    string$|&ZTXTHandle&|%TEXTresID%|#container$$¬
    font, size, style, mode, red, green, blue
Edit$(efID, -1) = ¬
    string$|&ZTXTHandle&|%TEXTresID%|#container$$¬
    font, size, style, mode, red, green, blue
Edit$(efID, selStart, selEnd) = ¬
    string$|&ZTXTHandle&|%TEXTresID%|#container$$¬
    font, size, style, mode, red, green, blue

```

Revision:

January 11, 2001 (Release 4)

Description:

This statement replaces all or a portion of the text in the edit field whose ID is *efID* in the current output window. *string\$* is any string expression. *ZTXTHandle&* is a handle to a “ZTXT” block (a “ZTXT” block contains text and (optionally) style information. See the `Get Field` statement for more information.) *TEXTresID%* is the resource ID number of a “TEXT” resource in an open resource file. A container is a variable that can hold up to 2 gigabytes of information. In the case of edit fields, the information is truncated at 32,767 characters which is the maximum number of characters allowed in an edit field.

If the field is a multistyled field, and you use the *&ZTXTHandle&* syntax, then any style information found in the “ZTXT” block is applied to the field’s text.

If a single parameter is used, the `Edit$` statement attempts to replace as much of the edit field text as will fit (up to 32K).

If a second parameter of -1 is added, the function replaces the current selection in whole or part depending on the size of the source variable. A second parameter that consists of a numeric value greater than or equal to zero specifies the line number that is to be replaced.

Where three parameters are used, the second and third values specify the starting and ending points of text to be replaced.

In all cases, FB³ insures that field is not overflowed.

When the style parameter is used, the runtime toggles specific style attributes instead of setting them. For instance, the bold style may be toggled on then off again by using the same style parameter of `_boldbit%`. If you wish to reset the selection to a particular style regardless of its current state, then begin by using a style parameter of zero before resetting the desired bits.

Inserted text may take on specific font characteristics as described by new parameters. This feature is available starting with Release 4. You may describe a font (by number), a size, style, and text mode. RGB colors are also definable using the red, green and blue parameters. Any of these parameters may be eliminated as long as a comma is retained as a place holder. The following example places "Mississippi" in the third line of the field using red text in a Monaco font.

```
Edit$(4,3) = "Mississippi",_monaco,,,-1
```

Example:

```
Edit$(1) = "555-2467"
Edit$(2) = &myZTXThndle&
Edit$(3) = %-350
Edit$(4,-1) = ReplaceCurrSel$
Edit$(5,12) = ReplaceLine12$
Edit$(6,10,20) = "Replace characters 10 thru 20 with this."
Edit$(7,-1) = ReplaceCurrSel$,newFont,newSize,newStyle
Edit$(8) = #myBigContainer$$
```

Colors

Colors can be specified either as `red,[green,[blue]]` or `@rgb`, where `rgb` is an `RGBColor` record. Note: the runtime distinguishes between a red value and an `rgb` address by taking any value from 0 to 65535 to mean "red", with any other value meaning "address of `rgb` record". Thus you cannot use a negative red values and must use the unsigned integer equivalent, for example 45536 not -20000.

See Also:

```
Edit$ function; Get Field; Edit Field; Read Field; Window(_efNum)
```

Edit Field

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax (Standard BASIC):

```

Edit Field [#]idExpr [, [text] [, [rect] [, [type] [, efClass]]]]
Edit Field 0

```

Syntax (Appearance Manager):

```

Edit Field [#]idExpr [, [text] [, [rect] [, [type] [, ↵
                                     [efClass] [, keyFilterProc]]]]
Edit Field -idExpr
Edit Field 0

```

Revision:

February, 2002 (Release 6)

Description:

Use this statement to perform any of the following actions in the current output window:

- Create a new edit field
- Activate an existing editable (non-static) edit field
- Modify the characteristics of an existing edit field
- Deactivate all editable fields in the window
- Disable a field (Appearance Manager only)

In the new Appearance Manager runtime, edit fields are actually buttons. FB creates special controls called user panes that allow styled text to be present. It is important to note that items created with the `Edit Field` statement are different from those created with the `Appearance Button` statement. Text items made with `Appearance Button` do not accommodate multiple runs of styled text, proper scrolling, and other necessary items. Text items created with the `Edit Field` statement have most of the features found in Standard BASIC edit fields with the added advantages of Appearance Manager compliance.

To disable an Appearance Manager field, use `Edit Field` with a negative value. To gray out field #22, you would use the statement `Edit Field -22`.

In FB³, an edit field can be either static or non-static. A static edit field contains text that is used for display purposes only; the user cannot edit the contents of such a field. A non-static field is editable; the user can use mouse and keyboard commands to edit the field's contents. It is possible for the program to change a field's type from static to non-static (or vice-versa); this is sometimes useful when you want to temporarily inhibit the user from editing some text. The Appearance Manager adds the ability to create "copy only" fields in which text may be selected and copied to the desk scrap but may not be edited.

Key Filter Procs

It is possible to filter key presses directed at fields in both the Appearance Manager and Standard BASIC. In Standard BASIC, the process involves waiting for a key press (see `On Edit`), checking the target for that key press (see `Window(_efNum)` function) and determining the appropriateness of that key.

The Appearance Manager has the ability to direct every keypress destined for a target field to a special procedure which you establish in your program and point to at the time the field is created. If you do not set up such a filter, the standard `On Edit` vector can still be used. The following fully functional example creates a simple numeric filter that limits field entry to the digits zero through nine.

```

Local Fn numeralFilter
  Dim k As str15
  k = TEKey$
  Long If k >= "0" And k <= "9"
    TEKey$ = k
  Xelse
    Select asc( k )
      Case < 32 : Rem allow for control keys
        TEKey$ = k
      Case Else
        Beep
    End Select
  End If
End Fn

Dim @ filterFN As Pointer
Dim r As Rect

Window 1
SetRect( r, 10, 10, 450, 60 )
filterFN = @Fn numeralFilter
Edit Field 1, "Numbers Only", @r,,, filterFN

Do
  HandleEvents
Until 0

```

A non-static (editable) field can be either active or inactive. In the frontmost window, an active field contains the blinking insertion point or a highlighted selection; it is the field whose contents the user is currently editing. At most one field in the window can be active at any given time; whenever a field becomes active, all other editable fields in the window become inactive. It is also possible to inactivate all of the editable fields in a window.

If your window contains an active edit field, then your program should call `HandleEvents` periodically (it's a good idea for your program to do this in any case). This will allow the user's keypresses and mouse events to be transmitted to the active field as appropriate, and will cause the field's contents to be updated correspondingly. Your program can use the `Edit$` function or the `Get Field` statement to check the contents of a field at any time. If you need to look at each keypress as the user enters it, designate an edit-handling routine with the `On Edit` statement, and use the `TEKey$` function to trap the user's keypresses.

FB³ automatically refreshes (redraws) both static and non-static edit fields as necessary, unless the window's `_noAutoClip` feature is set.

To create a new edit field: Specify a positive or negative number in `idExpr`, such that `Abs(idExpr)` is different from the ID numbers of all other existing edit fields or picture fields in the current window. The `rect` parameter is required in this case. If `idExpr` is positive, then a “non-styled” edit field is created, and is assigned an ID number equal to `idExpr`. If `idExpr` is negative, then a “multistyled” edit field is created, and is assigned an ID number equal to `Abs(idExpr)`. A “non-styled” edit field adopts the text characteristics (font family, size, style and color) that were in effect for the window at the time the field was created, and all the text in that field has those characteristics. A “multistyled” edit field can contain different pieces of text which each have different text characteristics.

Note: When you create a new non-static edit field, the new field becomes the active field in that window.

When you create a new edit field, any parameters that you omit have these default values:

- `text` defaults to a null string (i.e., the field is empty)
- `type` defaults to `_framedNoCR` (this is one of the editable types)
- `efClass` defaults to zero, which is one of the left-justified classes.

To activate an existing non-static edit field: Specify the ID number of the existing field in `idExpr`. You don't need to specify any other parameters, unless you also wish to alter some of the field's characteristics.

To modify the characteristics of an existing edit field: Specify the ID number of the existing field in `idExpr`, and specify one or more of the other parameters. Any parameter that you omit won't have its characteristic changed. Note that if the field is editable (non-static), this command will also make it the active field.

If the only thing you want to change about the field is its contents, then you can alternatively use the `Edit$` statement. The `Edit$` statement will not make the field active.

To inactivate all edit fields in the window: Use the `Edit Field 0` syntax.

To make a static field editable: You can specify any of the editable types in the `type` parameter.

Note: Changing a static field to an editable field also makes the field active.

To make an editable field static: First, deactivate the field (either by executing `Edit Field 0` or by activating a different field), and then specify any of the static types in the `type` parameter.

The following sections explain the use of the various parameters.

text

This parameter can be specified in any of the following forms:

- `string$` -- Any string expression.
- `&ZTXTHandle&` -- An “&” symbol followed by a handle to a “ZTXT” structure.
- `%TEXTresID%` -- A “%” symbol followed by the resource ID number of a “TEXT” resource.
- `#container$$` -- A “#” symbol followed by a container variable indicates that the contents of the container should be copied to the field. If the container has a length that is greater than 32,767, then only the first 32,767 bytes are copied to the field.

The field’s contents are completely replaced by the specified text. The various forms of this parameter are interpreted the same way as in the `Edit$` statement; see the `Edit$` statement for more information.

rect

This parameter specifies the “view rectangle” for the field. No text will be drawn outside of this rectangle. This parameter can be specified in either of the following forms:

- `(x1%, y1%) - (x2%, y2%)` These coordinates specify two diagonally opposite corners of the rectangle.
- `@rect` This is interpreted as the address of a standard 8-byte “rect” structure

type

This is an integer which specifies several characteristics about the field.

Please note that not all of the old FBII style field types are available for the Appearance Manager Runtime. The following list shows all of the types that are *not* available for Appearance projects:

```
_statFramedGray
_statNoFramedGray
_statFramedInvert
_statNoFramedInvert
_statFramedInvert + _hilite
_statNoFramedInvert + _hilite
```

The following 24 types only are supported, (each in two forms, with and without `_usePlainFrame`, making 48 altogether):

<code>_copyOnlyFramed</code>	<code>_copyOnlyFramed_auToGray</code>
<code>_copyOnlyNoFramed</code>	<code>_copyOnlyNoFramed_auToGray</code>
<code>_framedNoCR</code>	<code>_framedNoCR_auToGray</code>
<code>_framed</code>	<code>_framed_auToGray</code>
<code>_noFramedNoCR</code>	<code>_noFramedNoCR_auToGray</code>
<code>_noFramed</code>	<code>_noFramed_auToGray</code>
<code>_framedNoCR_noDrawFocus</code>	<code>_framedNoCR_auToGray_noDrawFocus</code>
<code>_framed_noDrawFocus</code>	<code>_framed_auToGray_noDrawFocus</code>
<code>_noFramedNoCR_noDrawFocus</code>	<code>_noFramedNoCR_auToGray_noDrawFocus</code>
<code>_noFramed_noDrawFocus</code>	<code>_noFramed_auToGray_noDrawFocus</code>
<code>_statFramed</code>	<code>_statFramed_noAuToGray</code>
<code>_statNoFramed</code>	<code>_statNoFramed_noAuToGray</code>

Editable (non-static) types:

<i>Parameter</i>	<i>Value</i>	<i>Description</i>
<code>_framedNoCR</code>	1	Frame is drawn. “Return” key does not advance line, but generates an <code>efReturn</code> event. This is the default type.
<code>_framed</code>	2	Frame is drawn. “Return” key advances insertion point to next line, does not generate an <code>efReturn</code> event.
<code>_noFramedNoCR</code>	3	Like <code>_framedNoCR</code> , but no frame is drawn.
<code>_noFramed</code>	4	Like <code>_framed</code> , but no frame is drawn.
<code>_statFramed</code>	5	Frame is drawn.
<code>_statNoFramed</code>	7	No frame is drawn.
<code>_statFramedGray</code>	9	Frame is drawn; frame and text are dimmed.
<code>_statNoFramedGray</code>	11	No frame is drawn; text is dimmed.
<code>_statFramedInvert</code>	13	Frame is drawn; text and background colors are inverted.
<code>_statNoFramedInvert</code>	15	No frame is drawn; text and background colors are inverted.
<code>_statFramedInvert</code> + <code>_hilite</code>	29	Frame is drawn; text and background are highlighted using system highlight colors.
<code>_statNoFramedInvert</code> + <code>_hilite</code>	31	No frame is drawn; text and background are highlighted using system highlight colors.
<code>_noDrawFocus</code> (Runtime Appearance)	256	Use this option if you don't want the focus rectangle outlining the active field.
<code>_noAutoGray</code> (Runtime Appearance)	512	When the window goes to the background, text in this field will not be grayed
<code>_autoGray</code> (Runtime Appearance)	1024	Text is automatically grayed when the window goes to the background.
<code>_copyOnlyFramed</code> (Runtime Appearance)	2048	The user may select and copy text from this field but may not edit the text.
<code>_copyOnlyNoFramed</code> (Runtime Appearance)	2051	This is the same as <code>_copyOnlyFramed</code> without the frame.
<code>_usePlainFrame</code> (Runtime Appearance)	4096	When added to a <code>_framed</code> or <code>_framedNoCR</code> type, gives an old-fashioned but crisp rectangular frame instead of the trendy fuzz obtained with <code>DrawThemeEditTextFrame</code> . The runtime currently requires a frame/CR constant to be supplied as well as the special <code>_usePlainFrame</code> . Thus you will have to specify: <code>_framedNoCR_usePlainFrame</code> or <code>_framed_usePlainFrame</code>

You can also add any combination of the following constants to the *type* parameter to achieve different effects in appearance (standard BASIC only):

<i>Constant</i>	<i>Value</i>	<i>Description</i>
<code>_round</code>	32	<code>_round</code> causes the frame or background highlighting to be oval-shaped. <code>_rounder</code> and <code>_roundest</code> produce different rounded-rectangle shapes.
<code>_rounder</code>	64	
<code>_roundest</code>	96	
(= <code>_round</code> + <code>_rounder</code>)		
<code>_boldBox</code>	128	For framed edit fields, draws a thicker frame.

efClass

This parameter must be within the range 0 through 255 (0 through 536,870,912 in the Appearance Manager). It serves two purposes:

- It determines how the lines of text will be positioned in the field (i.e., left-justified, right-justified or centered);
- For editable fields, it assigns a user-defined “class number” to the field. The number you specify will subsequently be returned by the `Window(_efClass)` function when the field is active.

If *efClass* is zero, then the text will be left-justified. Otherwise, text justification is determined by the low-order two bits in *efClass* (given by *efClass* Mod 4), as follows:

<i>If efClass Mod 4 is:</i>	<i>Then the text is:</i>
Zero	Right justified.
<code>_leftJust</code> (1)	Left justified.
<code>_centerJust</code> (2)	Centered.
<code>_rightJust</code> (3)	Right justified.

If you just want to set the field’s text justification, and you don’t care about its “class,” then the easiest thing to do is just to set *efClass* to one of the constants `_leftJust`, `_centerJust` or `_rightJust`.

Note:

An edit field cannot contain more than 32,767 characters of text.

The `Edit Field` statement is not the only means by which an inactive field can become active. If the user clicks on an inactive editable field, FB^3 automatically activates the field the next time your program executes the `HandleEvents` statement.

See the `Scroll Button` statement to learn how create a scroll bar that scrolls the text in a multistyled edit field.

See Also:

```
Def <just>Box; Def ShadowBox; HandleEvents; Edit; Edit Text;
Edit$ statement; Edit$ function; Get Field; Read Field; Picture Field;
Scroll Button
```


Edit Field Close

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Edit Field Close [#] fieldID%|&
```

Description:

This statement removes the specified field from the current output window. *fieldID%* (standard BASIC) or *fieldID&* (Appearance Runtime) can refer either to an edit field or a picture field. Note that all fields in a window are closed automatically whenever the window is closed. The contents of the field are no longer accessible by the `Edit$` function or the `Get Field` statement after the field has been closed.

See Also:

```
Edit Field; Edit$ function; Get Field; Picture Field
```

Edit Menu

statement

✓ *Appearance*

✓ *Standard*

✗ *Console*

Syntax:

```
Edit Menu menuParm
```

Description:

This statement builds an edit menu (see illustration) with the following characteristics:

- If an edit field is active in the current output window, then the menu's Cut, Copy, Paste and Clear options move text between the edit field and the "TextEdit Scrap" (a private clipboard area) in the standard ways. If the active field is a multistyled field, then style information is also moved.
- If a multistyled edit field is active in your application's output window, then FB^3 exchanges text and style information between the TextEdit scrap and the clipboard whenever your application is moved to the front or to the back (this allows you to cut & paste text to and from other applications).

If *menuParm* is greater than zero, then it's interpreted as a menu ID number, and the menu shown in the illustration is put into the specified position (almost always "2") in the menu bar.

If *menuParm* is negative, then FB^3 looks for a "Menu" resource whose resource ID number is *Abs(menuParm)*, and uses that resource to build the menu. This is useful in case you need to internationalize your program, or you need to have extra options listed in the menu. You should not change the positions of the Cut, Copy, Paste or Clear options in your "MENU" resource; they must be in the same positions as in the "standard" Edit menu (in the illustration) in order for FB^3 to interpret them correctly.

If an edit field is active in the current window, then selecting the Edit menu's Cut, Copy, Paste and Clear options will not generate FB^3 Menu events (FB^3 will handle these options transparently to your program). If the user selects any other item from the Edit menu, a Menu event will be generated.

If there is no active edit field in the current window, then all items in the Edit menu will generate Menu events.

Edit	
Can't Undo	⌘Z
Cut	⌘X
Copy	⌘C
Paste	⌘V
Clear	

If you wish to turn off automatic handling of Edit menu events, use:

```
Edit Menu 0.
```

See Also:

Menu function; Menu statement; On Menu; HandleEvents; Edit Field

Edit Text

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax (Standard BASIC):

```

Edit Text [#optionalFieldNum%,][font]~
        [, [size][, [style][, [mode][, red, green, blue]]]]

```

Syntax (Appearance Runtime):

```

Edit Text [#optionalFieldNum%,][font]~
        [, [size][, [style][, [mode][, ForeRGB][, backRGB]]]]

```

Revision:

February, 2002 (Release 6)

Description:

If the active edit field or the field specified by *#optionalFieldNum%* in the current output window is a multistyled edit field, then `Edit Text` applies the specified text characteristics to any text which is currently selected. Also, any new text which is subsequently inserted at the current insertion point or within the current selection range will be displayed using the specified text characteristics.

If the active field (or the field specified by *#optionalFieldNum%*) is not a multistyled field, then `Edit Text` causes the specified text characteristics (except color) to be applied to all the text in the field.

The parameters have the following meanings. If you omit any parameters, the corresponding text characteristics won't be altered.

The leading "#" symbol is required if the *optionalFieldNum* parameter is used (e.g. `Edit Text #3, _geneva, 12, _boldBit%, foreRGB, backRGB`).

<i>#optionalFieldNum</i>	field number that should accept the font changes. When omitted, the active edit field is used.
<i>font</i>	font family ID
<i>size</i>	font size, in points
<i>style</i>	bold, italic, underline, etc. See the <code>Text</code> statement for more information.
<i>mode</i>	This parameter is currently ignored.
<i>red, green, blue</i> (<i>Standard BASIC</i>)	Color components. Each component can range from 0 (darkest) to 65,535 (lightest)
<i>ForeRGB</i> (<i>Appearance</i>)	An RGB color record for the text color which could be dimensioned as: <code>Dim ForeRGB As RGBColor</code>
<i>backRGB</i> (<i>Appearance</i>)	An RGB color record for the background color which could be dimensioned as: <code>Dim backRGB As RGBColor</code>

Example:

CD Example: Edit Text.bas

See Also:`Text; SetSelect; Edit Field`

Eject statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Eject driveExpr
```

Description:

Use this statement to eject a floppy disk or other removable media, and optionally to unmount the volume.

If *driveExpr* is a positive number, then the statement ejects the volume in the drive whose Drive ID number is *driveExpr*, but does not unmount the volume.

If *driveExpr* is negative, then the statement ejects the volume in the drive whose Drive ID number is *Abs (driveExpr)*, unless the volume is in use by some application. If the volume is successfully ejected, then it is also unmounted.

Unmounting a volume causes its icon to disappear from the desktop, and causes its volume reference number to become invalid. If you eject a volume without unmounting it, then its icon becomes dimmed, but does not disappear. In this case, the system still “remembers” the volume, and will automatically ask you to re-insert it if any applications need to access it.

Every volume mounted on the desktop has a Drive ID number, including network volumes. If you specify a network volume’s Drive ID number (or its negative) in the `Eject` statement, and the volume is not in use by any application on your machine, then the volume will be unmounted, just as if you had dragged its icon to the Trash.

Example:

The Drive ID number of the internal floppy drive is always 1. `Eject 1` ejects the floppy disk but does not unmount it. `Eject -1` ejects the disk (if it’s not in use) and also unmounts it.

Note:

You can use the `HGetVInfo` Toolbox function to find the Drive ID numbers of all currently mounted volumes.

See Also:

Dialog function (`_evDiskInsert event`)

End statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**`End`**Description:**

This statement calls your stop-event handling routine (if you've designated one using the `On Stop` statement), then closes all open files and ports, disposes of all handles and pointers, releases all resources, and stops execution of the program.

Console Behavior:

The Text Window (and the Graphics Window, if any) remain open. The application does not quit until you select "Quit" from the "File" menu, or click the close box in the Text Window.

Note:

FB^3 always inserts an implicit `End` statement following the last executable line in your program. You don't need to explicitly include an `End` statement unless you want the program to end somewhere before that last line is reached.

See Also:

`Stop`; `Break On/Off`; `System`; `ShutDown`

End Enum**statement**

See the `Begin Enum` statement.

End Fn**statement**

See the `Local Fn` and `Long Fn` statements.

End Globals**statement**

See the `Begin Globals` statement.

End If**statement**

See the `Long If` statement.

End Record**statement**

See the `Begin Record` statement.

End Select**statement**

See the `Select` statement.

EndAssem**statement**

See the `BeginAssem` statement.

EnterProc

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax 1:

```

"statementLabel"
EnterProc [(intVar1 [,intVar2 ...])]
    [statementBlock]
ExitProc [= intVar3]

```

Syntax 2:

```

"statementLabel"
EnterProc Fn fnName[(intVar1 [,intVar2 ...])] [= type]
    [statementBlock]
ExitProc [= returnValue]

```

Description:

The `EnterProc` statement marks the beginning of a procedure block, which must end with an `ExitProc` statement. A procedure block is used to implement a function that uses the Pascal calling convention, which is a certain way that the function manipulates the stack. The most common use for this kind of function is to implement Toolbox “callback” procedures (often called “application-defined routines” in Inside Macintosh). Such callback procedures must use the Pascal calling convention.

An `EnterProc` procedure block can only appear within the “main” scope of the program; not within a `Local Fn`. Each of the `intVar`’s must be either a short (2-byte) integer variable or a long (4-byte) integer variable, or a `Pointer` or `Handle` variable. The `statementBlock` can contain any collection of `FB^3` statements except another `EnterProc...ExitProc` block (i.e., procedure blocks cannot be nested). The scope of all the variables referenced in the procedure block is either global (if the variable was declared earlier in a `Begin Globals...End Globals` block) or local to the procedure block itself.

A procedure block is called when some caller (typically a Toolbox routine) refers to the procedure’s address in memory; your program can use the `Proc "statementLabel"` function to get that address. Typically, your program would pass that address to the Toolbox routine, to identify the procedure block as a “callback” procedure. When the procedure block is called, `intVar1`, `intVar2`, etc. are assigned values which are passed by the caller. When the `ExitProc` statement is reached, control is returned back to the caller, and if `intVar3` (or `returnValue`) was specified, its value is returned to the caller.

It's important that you know the number and types of parameters (if any) that the caller expects to pass to your procedure block, and that you set up the *intVar1*, *intVar2* (etc.) parameters in the `EnterProc` statement accordingly. Likewise, you should know whether the caller expects your procedure block to return a value and, if so, whether the caller expects a 2-byte or a 4-byte value. If you use Syntax 1, you should specify a variable of the appropriate size in *intVar3* if the caller expects a return value; otherwise you should omit *intVar3*. If you use Syntax 2, you should specify the name of a 2-byte or 4-byte type (such as `Int` or `Pointer`) in the *type* parameter, and specify a *returnValue* expression of that type, if the caller expects a return value; otherwise, you should omit the *type* and the *returnValue*. If you do not set up the input parameters and the return parameters correctly in your `EnterProc` and `ExitProc` statements, then your program is likely to crash when the procedure block is called. See Appendix C: *Data Types and Data Representation*, for a list of valid 2-byte and 4-byte variable types.

If you use Syntax 2, you can also execute a procedure block by using the `Fn <userFunction>` statement, although there is generally little reason to do this (it's usually better to use a `Local Fn` when you want to create a procedure that will be explicitly called by your program).

Note:

When your program starts up, it will not automatically “skip over” the lines in a procedure block, the way it skips over the lines in a `Local Fn`. In order to prevent your procedure block from being executed inadvertently, you should either branch around the procedure block using `Goto`, like this:

```
Goto "jump"
"myProc1"
EnterProc(var&)
:
ExitProc
"myProc2"
EnterProc(var1%, var2&)
:
ExitProc = result&
"jump"
```

or you should place the procedure block after an `End` statement, like this:

```
'[main part of program here]
End
'=====  
Procedures  
=====  
"myProc1"  
EnterProc(var&)  
:  
ExitProc  
"myProc2"  
EnterProc(var1%, var2&)  
:  
ExitProc = result&
```

See Also:

Appendix C: *Data Types and Data Representation*

Eof**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
endReached = Eof(fileID)
```

Description:

This function returns `_zTrue` if the “file mark” associated with the specified open file is positioned at the end of the file. The “file mark” is an internal pointer which is used by all operations which read from or write to the file; it indicates where in the file the next data should be read from or written to. You can move the file mark explicitly using the `Record` statement; the file mark is also advanced automatically every time you do a read or write operation. Typically, you use the `Eof` function when reading data sequentially, to determine when no more data can be read from the file.

Example:

CD Example: Eof.bas

Note:

If your program attempts to read data past the end of the file, `FB^3` returns an `_endOfFile` error to your program.

See Also:

Error function; Error statement; On Error Fn; On Error Gsub; SysError;
Open; Close; Record; Rec; Pos; Lof

Erf# & Erfc#**functions**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
error# = Fn Erf#(z#)
complimentaryError# = Fn Erfc#(z#)
```

Description:

`Erf` is shorthand for error function. `Erfc` stands for complementary error function. The error function `Erf[z]` is the integral of the Gaussian distribution given by

$$\text{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$$

The complementary error function `ERFC[z]` is

$$\text{erfc}(z) = 1 - \text{erf}(z).$$

Note:

The error functions are located in the file named “Subs Float Addns.Incl” (Path: FB Extensions/Compiler/Headers). To provide access to these functions you must use the following statement in your program.

```
Include "Subs Float Addns.Incl"
```

FB^3 will automatically locate the file and compile it with your project. The `Erf` and `Erfc` functions are provided as local functions and expect a double precision (#) value as the incoming parameter. The return value is also a double precision (#) number.

Error

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
errorInfo% = Error
```

Description:

This function returns information about the most recent disk i/o error encountered in your program. It relates only to i/o errors in the FB^3 runtime; it does not include string errors, numeric errors, or error codes returned by MacOS Toolbox functions.

The `Error` function returns a 2-byte integer. The low-order byte contains an error code. Usually, the error involves some file operation; when this is the case, the high-order byte contains the file ID number of the relevant file. You can retrieve both of these pieces of information as follows:

```
errorCode% = Error And &FF
fileID%     = Error >> 8
```

`errorCode%` will be set to one of the following values:

```
_noErr           (0)
_endOfFile       (1)
_diskFull        (2)
_fileNotFound    (3)
_fileNotOpen     (4)
_badFileName     (5)
_badFileNum      (6)
_writeOnly       (7)
_readOnly        (8)
_posErr          (9)
_diskErr         (10)
_systemErr       (11)
_openTypeErr     (12)
```

Note that these kinds of errors will normally cause your program to stop executing. If you want to trap and handle these errors within your program, you must use the `On Error Fn` and `On Error End` statements.

If the error code comes back as `_systemErr` then you need to check the value of `SysError`. You may also check the value of `SysError` at any time to learn the toolbox result code.

Note:

After retrieving the error information with the `Error` function, you should reset FutureBASIC's internal error register by executing the `Error = _noErr` statement.

See Also:

`Error` statement; `On Error Fn`; `On Error Gsub`; `SysError`

Error

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Error = _noErr
```

Description:

This statement clears FutureBASIC’s internal error register after an error has occurred. If your program has an error-handling function (defined using the `On Error Fn` statement), then you should execute this statement within that function, after you have used the `Error` function to retrieve information about the error.

See Also:

```
Error function; On Error Fn; On Error Gsub; SysError
```

Event**function**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
eventRecPtr& = Event
```

Description:

This function returns a pointer to a system event record as defined in the Event Manager chapter of Inside Macintosh: *Macintosh Toolbox Essentials*. If your program has designated a system event-handling routine using the `On Event` statement, then your routine should check the contents of this record every time the routine is called. The record will contain a description of the event that triggered the call to your system event-handling routine.

If you store the record pointer into a long integer or `Pointer` variable (e.g., `eventRecPtr& = Event`), then you can examine the individual fields of the event record as follows:

```
type% = eventRecPtr&.evtNum%
```

This represents the event type, which will be one of the following constants:

<i>Event type</i>	<i>Value</i>	<i>Description</i>
<code>_nullEvt</code>	0	No event occurred
<code>_mButDwnEvt</code>	1	mouse button pressed
<code>_mButUpEvt</code>	2	mouse button released
<code>_keyDwnEvt</code>	3	key pressed
<code>_keyUpEvt</code>	4	key released
<code>_auToKeyEvt</code>	5	key repeatedly held down
<code>_updatEvt</code>	6	window needs updating
<code>_diskInsertEvt</code>	7	disk inserted
<code>_activateEvt</code>	8	window was brought to front or moved to back (see the <code>evtMeta%</code> field to determine which)
<code>_osEvt</code>	15	operating system events (suspend, resume, mouse moved)
<code>_kHighLevelEvent</code>	23	high-level events (includes Apple Events)

```
message& = eventRecPtr&.evtMessage&
```

This contains additional information about the event, which varies depending on the type of event that occurred.

type%	message&
_keyDwnEvt, _keyUpEvt, _auToKeyEvt	bits 0-7 = ASCII char; bits 8-15 = key code; for ADB keyboards, bits 16-23 = keyboard's ADB address.
_updateEvt, _activateEvt	pointer to the window
_diskInsertEvt	bits 0-15 = drive ID; bits 16-31 = error code
_osEvt	high-order byte = 1 for suspend/resume events; high-order byte = 250 for mouse-moved event. For suspend/resume events: bit 0 = 0 for suspend; bit 0 = 1 for resume. For resume event: bit 1 = 1 if clipboard was changed.
_kHighLevelEvent	Class of events to which the high level event belongs. This is used with the _evtMouse field to identify the specific type of high-level event received.

```
ticks& = eventRecPtr&.evtTicks&
```

This gives the tickcount value (time since system startup, in ticks) when the event occurred. You can compare this against the current value of `Fn TickCount`, to determine how long ago the event occurred.

```
mousePt;4 = eventRecPtr& + _evtMouse
```

For low level events (i.e., all types except `_kHighLevelEvent`), this gives the mouse cursor location, in global coordinates, at the time the event occurred. `mousePt` should be Dim'ed as a 4-byte record.

```
highLevelEvtID& = eventRecPtr&.evtMouse&
```

For high-level events (i.e., of type `_kHighLevelEvent`), this gives the high-level Event ID, which together with `message&` identifies the type of high-level event.

```
modKeys% = eventRecPtr&.evtMeta%
```

Contains information about the state of the modifier keys (Control, Shift, etc.) and the mouse button, at the time the event occurred. For activate events, bit 0 = 1 if the window should be activated; bit 0 = 0 if the window should be deactivated. The state of the modifier keys and mouse button are determined by specific bit values in this short integer; see the `Event%` function for more information.

“Clearing” an Event

After you’ve finished looking at an event record and you exit your system event-handling routine, FB^3 examines that same event record to determine whether the event can be translated into one of the “FB^3 events” that your program typically detects using the `Dialog` function, the `Menu` function, etc. For example, if FB^3 sees an event record in which a `_mButDwn` event has occurred, it checks whether the mouse was clicked inside a button, and tracks the mouse in preparation for a possible `Dialog` event of type `_btnClick`.

If you’ve handled the event completely within your `On Event` routine, and you don’t want FB^3 to do anything else with the event record afterwards, then you should set the event type to `_nullEvt` before you exit your `On Event` routine. You can do this as follows:

```
eventRecPtr&.evtNum% = _nullEvt
```

This will “fool” FB^3 into thinking that the event was a null event, and the record will subsequently be ignored.

See Also:

```
Event% function; On Event; HandleEvents
```


Event&**function**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**`eventInfo& = Event&`**Description:**

This function can be called within most kinds of event-handling routines, and returns extra information about the detected event. For most kinds of events, the `Event&` function returns the time (in ticks since system startup) at which the event occurred. However, if the event is a `_userDialog` event, then the `Event&` function returns the long-integer value that was associated with this event by the `Event&` statement. (A `_userDialog` event is generated when your program executes the `Dialog` statement.)

The `Event&` function cannot be used within a system event-handling routine (as designated by `On Event`), nor within an edit-key handling routine (as designated by `On Edit`). If used within these two types of event-handling routines, or if used outside of an event handling routine, the value returned by `Event&` is undefined.

Note:

To detect the time of a system event within a system event-handling routine (as designated by `On Event`), look at the `_evTicks` field of the event record. See the `Event` function for more information.

For “extended” events that start and end at two different times (such as selecting a menu item), the `Event&` function returns the time at which the event started (for example, the time when the mouse was first clicked on the menu bar--not the time at which the mouse was released).

See Also:

`Dialog statement`; `Event& statement`; `Event% statement/function`; `Event function`

Event%**function**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
eventInfo% = Event%
```

Description:

This function can be called within most kinds of event-handling routines, and returns extra information about the detected event. For most kinds of events, the `Event%` function returns the status of the mouse button and the “modifier keys” (Shift, Option, etc.) in effect at the time the event occurred. However, if the event is a `_userDialog` event, then the `Event%` function returns the short-integer value that was associated with this event by the `Event%` statement. (A `_userDialog` event is generated when your program executes the `Dialog` statement.)

The `Event%` function cannot be used within a system event-handling routine (as designated by `On Event`), nor within an edit-key handling routine (as designated by `On Edit`). If used within these two types of event-handling routines, or if used outside of an event-handling routine, the value returned by `Event%` is undefined.

Example:

The modifier keys include Caps Lock, Shift, Control, Command and Option. The state of each of these keys (and of the mouse button) is represented by a separate bit in the value returned by `Event%`. In the case of the modifier keys, the bit is set if the key was down. In the case of the mouse button (bit 7), the bit is set if the button was up. You can use the following constants to test these bits:

```
_btnState      = 7
_cmdKey        = 8
_shiftKey      = 9
_alphaLock     = 10
_optionKey     = 11
_controlKey    = 12
```

For example, use this statement to determine if the Caps Lock key was down at the time the event occurred:

```
CapsLockDown% = (Event% And _alphaLock%) <> 0
```

(Note the “%” symbol added to the end of the constant name. The expression `_alphaLock%` is equivalent to the expression `Bit(_alphaLock)`. See the `Bit` function for more information.)

To detect the state of the modifier keys within a system event-handling routine (as designated by `On Event`), look at the `_evtMeta` field of the event record. See the `Event` function for more information.

See Also:

`Dialog` statement; `Event&` statement/function; `Event%` statement; `Event` function

Event<%l&> statements

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```

Event% = shortIntValue%
Event& = longIntValue%

```

Description:

Use these statements when you want to associate extra information with a particular `_userDialog` event (that is, an event that you post using the `Dialog` statement). Use the `Event%` statement to specify a short integer value, and use the `Event&` statement to specify a long integer value. When you subsequently execute a `Dialog` statement to post a `_userDialog` event, the value(s) that you specified will be associated with that particular posted event. The value(s) you specify can be anything that's relevant to your program, and can be read within your dialog event-handling routine, using the `Event%` and `Event&` functions, when your program detects the `_userDialog` event.

You can use either or both of these statements, depending on what information you need to associate with the event. When you use them they must be executed before the `Dialog` statement which actually posts the event. For example:

```

'Associate myShortValue% And myLongValue& with Next event
Event% = myShortValue%
Event& = myLongValue&
Dialog = myDialogID%

```

See Also:

`Dialog` statement; `Dialog` function; `Event%` function; `Event&` function

Exit structure statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Exit For
Exit Next
Exit Do
Exit Until
Exit While
Exit Wend
Exit Case
```

Revision:

May 30, 2000 (Release 3)

Description:

When used inside a `For/Next` loop, `While/Wend` conditional, `Do/Until` conditional, or in a `Select Case` structure, this statement causes the program to jump immediately to the line following the `Next`, `Wend`, `Until`, or `End Select` statement. This is useful when, you wish to break out of a loop because a certain condition has been met. `Exit` is a safe way to do it.

Example

```
For x = 1 To 10
  If Fn Button Then Exit For
Next x
```

See Also:

`For`

Exit Fn statement

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

Exit Fn

Description:

When used inside a `Local Fn` function, this statement causes the program to jump immediately to the `End Fn` statement. The function then exits, passing back the value (if any) specified in the `End Fn` statement. This is useful when, for example, you wish to break out of a loop and quit the function immediately; `Exit Fn` is a safer way to do this than using something like `Goto`.

Note:

You should not use the `Exit Fn` statement outside of a `Local Fn` function.

See Also:

`Local Fn; End Fn; Goto; Exit <label>`

Exit <label> statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Exit "label"
```

Revision:

January 2, 2001 (Release 4)

Description:

This statement causes the program to jump to the statement following the indicated label. Unlike the Goto statement, the Exit <label> statement makes sure that any loops, Long If blocks, Local Fn's, etc., that are being "jumped out" of get properly closed, so that the stack will be in a consistent state.

If an Exit "Label" is in the main program the "Label" must be inside the main. If an Exit "Label" is in a Local Fn the "Label" must be inside the same Local Fn. If an Exit "Label" is nested inside one or more Select [Case] statements the "Label" must be after the last End Select.

Any exceptions to the above conditions could result in substantial penalties or crashes.

See Also:

Goto; Exit Fn

ExitProc

statement

See the `EnterProc` statement.

Exp function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

result# = **Exp**(*expr*)

Description:

Returns the value of the transcendental number “e” raised to the power of *expr*. The number “e” is the base of natural logarithms, and is approximately equal to 2.718281828. The `Exp` function is used extensively in probability theory and in applied sciences.

`Exp` is the inverse of the `Log` function; that is: `Exp (Log (x))` equals `x`. `Exp` always returns a double-precision result.

See Also:

`Log`; `Log10`; `Log2`

FBCompareContainers

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
result& = Fn FBCompareContainers (a$$, b$$)
```

Revision:

June 12, 2000 (Release 3)

Description:

This function return a result that represents how container *a* compairs to container *b*. If the *result*& is zero, the containers are identical. A negative result (*-n*&) provides the character position at which container *a* was found to be less than container *b*. A positive result give the character position where container *a* became greater than container *b*.

<i>result</i> &	<i>Indicates</i>
Negative	container <i>a</i> < container <i>b</i>
Zero	container <i>a</i> = container <i>b</i>
Positive	container <i>a</i> > container <i>b</i>

Note:

With this function, containers are evaluated by ASCII (not numeric) values.

FBCompareHandles

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
result% = Fn FBCompareHandles (a%, b%)
```

Revision:

July 26, 2000 (Release 3)

Description:

This function returns a result representing a comparison of the contents of handle *a* with the contents of handle *b*. If *result%* is zero, the contents of the handles are identical. If *result%* is negative, *-result%* indicates the byte position at which handle *a* was found to be less than handle *b*. If *result%* is positive, it indicates the byte position at which handle *a* was found to be greater than handle *b*. The first position in the handle is byte number 1 (not zero). Two handles which differ from the very first byte will return a positive or negative 1 as a result.

<i>result%</i>	<i>Indicates</i>
Negative	handle <i>a</i> < handle <i>b</i>
Zero	handle <i>a</i> = handle <i>b</i>
Positive	handle <i>a</i> > handle <i>b</i>

FBGetControlRect**function**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
ignored = Fn FBGetControlRect(cHndl&, rect)
```

Revision:

August, 2002 (Release 7)

Description:

Before OS-X, we were able to extract the content rectangle of a button using the following code:

```
// does not work in OS-X
Dim @t,l,b,r
BlockMove Button&(_btnRefNum)+_ctrlrect,@t,8
```

Carbon programs do not offer the ability to peek into structures like control records. FB provides this utility function for use in any program so that you may extract the rectangle regardless of the current version of system software.

```
Dim @t,l,b,r // old style rectangle
Fn FBGetControlRect(Button&(_thebutton), t)
```

or...

```
Dim r As Rect // new style rectangle
Fn FBGetControlRect(Button&(_thebutton), r)
```

FBGetScreenRect**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
ignored = Fn FBGetScreenRect(rect)
```

Revision:

February, 2002 (Release 6)

Description:

Before OS-X, we were able to extract the content rectangle of a window using the following code:

```
// does not work in OS-X
Dim @t,l,b,r
BlockMove Window(_wndPointer)+portRect,@t,8
```

This no longer works because the window pointer and the grafport have been separated into different structures. You can substitute this simple function in your programs and it will work in all supported versions of the system software.

```
Dim @t,l,b,r // old style rectangle
Fn FBGetScreenRect(t)
```

or...

```
Dim r As Rect // new style rectangle
Fn FBGetScreenRect(r)
```

FBGetSystemName\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
name$ = Fn FBGetSystemName$(nameType)
```

Revision:

April, 2003 (Release 8)

Description:

This function returns the computer name or the user name according to the *nameType* parameter. *nameType* is one of the following constants:

`_FBComputerName`, `_FBLongUserName` or `_FBShortUserName`.

To make this routine available to your program, you must include the header file "Util_ComputerNames.Incl".

Example:

```
Include "Util_ComputerNames.Incl"

Print "  Computer Name: ";↵
      Fn FBGetSystemName$(_FBComputerName);""
Print "Long  User Name: ";↵
      Fn FBGetSystemName$(_FBLongUserName);""
Print "Short User Name: ";↵
      Fn FBGetSystemName$(_FBShortUserName);""

Do
  HandleEvents
Until 0
```

FBTestForLibrary**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
result& = Fn FBTestForLibrary("Library Name")
```

Revision:

July 28, 2000 (Release 3)

Description:

This function determines whether a library exists and returns a boolean result. Keep in mind that libraries often have one name that is visible in the Finder and a different name that is used for access. To determine the required name for `Library` statements, use a resource editor like Resorcerer and examine the `_"cfrg"` resource. The proper names for many common libraries can be found under the Help menu in the manual named "Mac Libraries."

Example:

```
Long If Fn FBTestForLibrary("ThreadsLib") = _false
  Stop " This program will not run without ↵
      the Thread Manager libraries."
Xelse
  Library "ThreadsLib"
  Rem Assign Toolboxes here
  Library
End If
```

See Also:

`Library`; `Toolbox`

Files\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
fileType$ = Files$
```

Description:

Returns the file type of the last file returned by the `Files$(_fOpen...)` function, as a 4-character string. If the user clicked “Cancel” in response to the last File Open dialog, or if the `Files$(_fOpen...)` function has never yet been executed, then the `Files$` function returns an empty (zero-length) string.

Note:

In some cases it’s useful to express the file type as a 4-byte long integer rather than as a string. Use the `Mki$` function and the `Cvi` function to convert between these two forms.

See Also:

```
Files$(_fOpen...); Mki$; Cvi
```

Files\$(_fFolder...)**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

folderName$ = Files$(_fFolder, [prompt$],, refNumVar%)
folderName$ = Files$(_FSSpecFolder, [prompt$],, FSSpec)

gFBUseNavServices = _zTrue|_false (See note below.)

```

Revision:

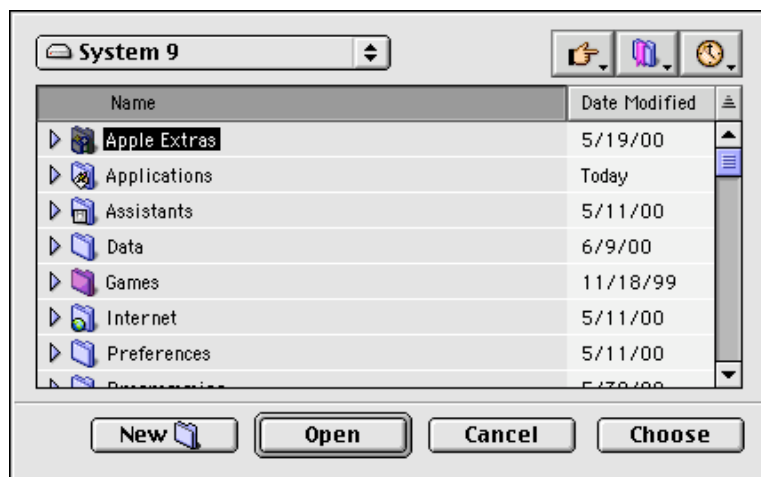
February, 2002 (Release 6)

Description:

This function prompts the user to select an existing folder. It does this by displaying the standard “Get File” dialog shown below (its appearance may be different on some systems). If the user selects a folder, then the folder’s name is returned in *folderName\$*, and a reference number for the folder is returned in *refNumVar%* (which must be a short integer variable). If the selected folder is in a volume’s root directory, the *refNumVar%* will return a volume reference number; otherwise, it will return a working directory reference number. If the user cancels the dialog, then the function returns an empty (zero-length) string, and it sets *refNumVar%* to zero.

The information contained in *folderName\$* is the correct name for the folder but is not of any real value except to determine that the user has selected Choose instead of Cancel. Only the volume reference number is necessary when using FB^3’s file handling routines.

If the global variable *gFBUseNavServices* is non-zero, FB switches to the more modern Navigation Services dialog. If the FSSpec version of the call is used, *gFBUseNavServices* is assumed to be *_zTrue*. For Carbon, the *gFBUseNavServices = _zTrue* statement is required.

**See Also:**

Appendix H: *File Spec Records*; *Files\$(_fOpen...);Open; Folder*

Files\$(_fOpen...)**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

fileName$ = Files$(_fOpen, [typeList$], [prompt$], refNumVar%)
fileName$ = Files$(_fOpenPreview, [typeList$], [prompt$], refNumVar%)
fileName$ = Files$(_FSSpecOpen, [typeList$], [prompt$], FSSpec)
fileName$ = Files$(_FSSpecOpenPreview, [typeList$], [prompt$], FSSpec)

gFBUseNavServices = _zTrue|_false

```

Revision:

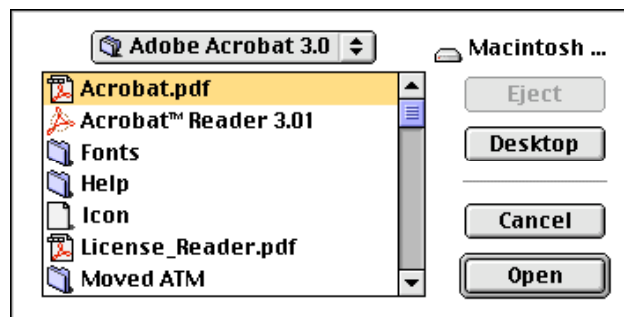
February, 2002 (Release 6)

Description:

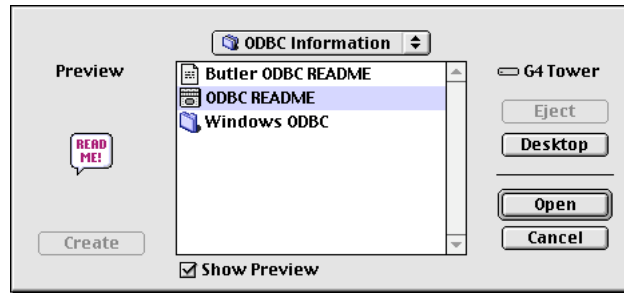
This function prompts the user to select an existing file. It does this by displaying the standard “Get File” dialog shown below (its appearance may be different on some systems). If the user selects a file, then the file’s name is returned in *fileName\$*, and a reference number for the file’s directory is returned in *refNumVar%* (which must be a short integer variable). If the selected file is in a volume’s root directory, the *refNumVar%* will return a volume reference number; otherwise, it will return a working directory reference number. If the user cancels the dialog, then the function returns an empty (zero-length) string, and it sets *refNumVar%* to zero.

If the global variable *gFBUseNavServices* is non-zero, FB switches to the more modern Navigation Services dialog. If the one of the FSSpec version of the call is used, *gFBUseNavServices* is assumed to be *_zTrue*.

You can limit the types of files that appear in the dialog by specifying up to four file types in *typeList\$*. For example, if you pass the string “TEXTPICT” in *typeList\$*, then only files of type “TEXT” and type “PICT” will be available for selection. If *typeList\$* is an empty string, or the parameter is omitted, then all file types will be available for selection.



If you use the optional `_fOpenPreview` (or `_FSSpecOpenPreview`) parameter, the standard files dialog may display a preview of the file currently selected.



Note:

The `Files$(_fOpen...)` function does not actually open the selected file. Use the `Open` statement if you need to open the file.

The reference number returned in `refNumVar%` is a temporary number, which is only valid until your program quits. You cannot use this same number to refer to this folder at a later date. If you need to keep track of a file's location over time, create and save an alias record for the file.

See Also:

`Files$`; `Files$(_fSave...)`; `Folder`; `Open`; [Appendix H: File Spec Records](#)

Files\$(_fSave...)**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

fileName$ = Files$(_fSave,[prompt$],[defaultName$],refNumVar%)
fileName$ = Files$(_FSSpecSave,[prompt$],[defaultName$],FSSpecRecord)

gFBUseNavServices = _zTrue|_false

```

Revision:

February, 2002 (Release 6)

Description:

This function prompts the user to provide a file name, and to select a folder where the file may be saved. It does this by displaying the standard “Put File” dialog shown below (its appearance may be different on some systems). If the user selects a name, then the name is returned in *fileName\$*, and a reference number for the selected directory is returned in *refNumVar%* (which must be a short integer variable). If the selected directory is a volume’s root directory, then *refNumVar%* will return a volume reference number; otherwise, it will return a working directory reference number. If the user cancels the dialog, then the function returns an empty (zero-length) string, and it sets *refNumVar%* to zero.

The string (if any) that you provide in *prompt\$* will appear as a one-line prompt in the dialog. The string (if any) that you provide in *defaultName\$* will appear initially in the file name edit field in the dialog.

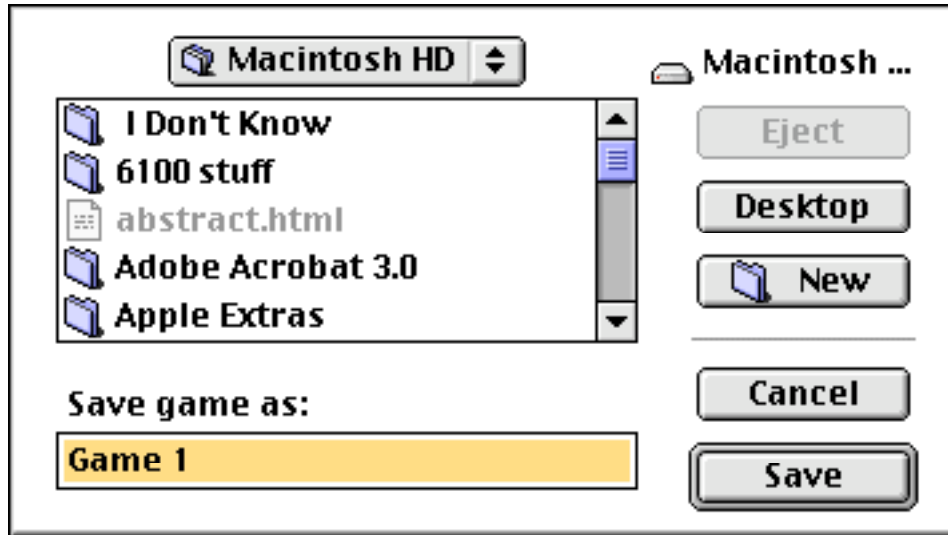
If the global variable *gFBUseNavServices* is non-zero, FB switches to the more modern Navigation Services dialog.. If the one of the FSSpec version of the call is used, *gFBUseNavServices* is assumed to be *_zTrue*.

Example:

The statement:

```
fileName$ = Files$(_fSave,"Save game as:","Game 1",refNum%)
```

will produce a dialog that looks something like this:

**Note:**

The `Files$(_fSave...)` function does not actually open or save a file. Use the `Open` statement if you need to open a file, and use output statements like `Print#` or `Write` to save information into it.

The reference number returned in `refNumVar%` is a temporary number, which is only valid until your program quits. You cannot use this same number to refer to this folder at a later date. If you need to keep track of a file's location over time, create and save an alias record for the file.

See Also:

`Files$`; `Files$(_fOpen...)`; `Folder`; `Open`; `Print#`; `Write`; Appendix H: *File Spec Records*

Files\$ <index>**function**

This function is obsolete. See `USR_SCANFOLDER` instead

Fill statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Fill h, v
```

Description:

This statement fills the area around pixel coordinates (h, v) (in the current output window) with the current color and pen pattern. All the contiguous pixels which have a color equal to the original color at (h, v) are included in the fill.

Example:

This little program paints a red ring:

```
Color = _zBlack
Circle 110, 110, 100
Circle 110, 110, 80
Color = _zRed
Fill 100, 15
```

Console Behavior:

When you use the Console runtime, `Fill` switches to the Graphics Window before executing.

See Also:

`Color`; `Circle`

FinderInfo

function

✓X Appearance

✓X Standard

✓X Console

Syntax:

Move all waiting items to arrays or simple variables

```
countVar = maxAcceptableentries
action = FinderInfo(countVar%, nameVar$, typeVar&, dirRefNumVar%)
```

Find out how many items are waiting to be picked up

```
countVar = 0
action = FinderInfo(countVar%, nameVar$, typeVar&, dirRefNumVar%)
```

Pick up an indexed item from the list

```
countVar = negativeIndex
action = FinderInfo(countVar%, nameVar$, typeVar&, dirRefNumVar%)
```

Gather a list of file spec records

```
countVar = maxAcceptableentries
action = FinderInfo(countVar%, @FSSpec[(array)], ↵
                    @OSType&[(array)], dirRefNumVar%)
```

Clear the list

Fn ClearFinderInfo

Description:

If the user launched your application by double-clicking a document icon, or by dragging document icon(s) to your application's icon, or by selecting document icon(s) and then selecting "Open" or "Print" from the Finder's "File" menu, then you can use the `FinderInfo` function to determine which document file(s) were involved, and whether they should be opened or printed.

You should call `FinderInfo` once, soon after your program starts. You should also check during null events to see if additional files have been added to the list. This can take place when an `_openDoc` event is sent from another application or when the user drags a file onto the icon of your running application.

- *action* – The result of the `FinderInfo` function is one of the following.

<i>action constant</i>	<i>Value</i>	<i>Description</i>
<code>_finderInfoOpen</code>	0	This file should be opened
<code>_finderInfoPrint</code>	1	This file should be printed
<code>_finderInfoErr</code>	2	An error occurred. One possible reason is that the program attempted to retrieve an indexed item that was out of range.

- *countVar* – This variable is used to send a value to and receive a result from `FinderInfo`

<i>countVar</i>	<i>Description</i>
< zero	An index into the list. The first item is -1, the next is -2, etc.
zero	The return value will be placed in <i>countVar</i> . It will be the total number of items available. This is reset to zero when you call <code>Fn ClearFinderInfo</code> .
> zero	In this case, <i>countVar</i> indicates the maximum number of entries that your program can accept. If you dimension an array to hold 10 elements, then the maximum would be 11 (10 + element zero). If you use 1, then the information can be placed in simple variables.

The parameters for `FinderInfo` are used to both send and receive values. In order to send a value of "1" for the count, you must first set the variable, then check it on return.


```
count% = 0
action = FinderInfo(count%, fName$, fType&, vRefNum%)
Print "There are" count% " files in the queue."
```

<i>nameVar\$\$</i>	<p>This must be a “short string” simple variable or array element.</p> <ul style="list-style-type: none"> • If you specify a maximum greater than 1 in <i>countVar%</i>, then you must specify an array element in <i>nameVar\$</i>, and the array must be dimensioned at exactly 31 characters per string. The names of the documents are returned into consecutive elements in the array, starting at the element you specify. • If you specify 1 in <i>countVar%</i>, then you can use a simple string variable for <i>nameVar\$</i>, dimensioned to at least 31 characters. The name of the document is returned in this variable.
<i>typeVar&</i>	<p>This must be a long integer simple variable or array element.</p> <ul style="list-style-type: none"> • If you specify a maximum greater than 1 in <i>countVar%</i>, then you must specify an array element in <i>typeVar&</i>. The 4-byte document type codes are returned into consecutive elements in the array, starting at the element you specify. • If you specify 1 in <i>countVar%</i>, then you can use a simple long integer variable for <i>typeVar&</i>. The type code of the document is returned in this variable.
<i>dirRefNumVar%</i>	<p>This must be a short integer simple variable or array element.</p> <ul style="list-style-type: none"> • If you specify a maximum greater than 1 in <i>countVar%</i>, then you must specify an array element in <i>dirRefNumVar%</i>. The directory reference numbers for the documents are returned into consecutive elements in the array, starting at the element you specify. • If you specify 1 in <i>countVar%</i>, then you can use a simple short integer variable for <i>dirRefNumVar%</i>. The directory reference number for the document is returned in this variable. <p>A document’s directory reference number indicates what directory the document is in. This will either be a volume reference number (in which case the document is in the volume’s root directory), or a working directory reference number.</p>

Note:

Before your application can support Finder-launched documents, you need to set up certain special resources (BNDL and FREF) in your application's resource fork. Also, `FinderInfo` will not work unless the "High-level event aware" flag is turned on in your application's `SIZE` resource #-1. See the "Finder Interface" chapter in *Inside Macintosh: Macintosh Toolbox Essentials*, for more information.

Fix**function**

 *Appearance* *Standard* *Console*

Syntax:

```
wholeNum# = Fix(expr#)
```

Description:

This function returns a whole number representation of *expr#* (it strips off digits to the right of the decimal point). Although `Fix` always returns an integer, the number it returns is considered to be a double-precision floating-point value.

See Also:

```
Frac; Int
```

FlushEvents	statement	
✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>

Syntax:

`FlushEvents`

Description:

This statement removes all pending events from the system event queue and from FutureBASIC's internal event queues. This is useful when you want to prevent your event-handling functions from processing any old events that may currently exist in the queue. `FlushEvents` does not inhibit future events from being put into the queue.

See Also:

`Event`; `On Event`

FlushWindowBuffer**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
FlushWindowBuffer [{wRef | {_FBAutoFlushOff | _FBAutoFlushOn}}]
```

Revision:

April, 2003 (Release 8)

Description:

Under OS-X, all drawing to a window is intercepted and stored ("buffered") by the Window Server. The Window Server normally transfers the drawing to the screen only when your program executes a `HandleEvents` statement. You can force an early update with `FlushWindowBuffer`.

If `wRef` is omitted, or is 0, the current output window is flushed by the OS-X Window Server. If `wRef` is non-zero, window `wRef` is flushed.

The default behavior of the FutureBASIC runtime is to flush the current output window each time a `Print` statement is performed. You can control that behavior:

`FlushWindowBuffer _FBAutoFlushOff` will turn off the automatic flushing. You still can force the flushing for a specific window with `FlushWindowBuffer wRef`.

`FlushWindowBuffer _FBAutoFlushOn` will restore the automatic flushing.

The `FlushWindowBuffer` command has no effect unless the program is running under OS-X.

Example:

```
// In this example, there is no HandleEvents,
// and so FlushWindowBuffer is needed to
// make the drawing visible under OS X.
```

```
Window 1
Plot 0,0 To 500,500
```

```
FlushWindowBuffer
```

```
Do
Until Fn Button // wait for mouse-down
```

Fn <toolbox>**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
result = Fn ToolboxFunctionName [modifiers] ↵
        [ ( [ { #addrExpr1 & | arg1 } [ , { #addrExpr2 & | arg2 } ... ] ] ) ]
```

Description:

This function executes a Toolbox function as defined in Inside Macintosh. A Toolbox function (as opposed to a Toolbox procedure) returns a value.

ToolboxFunctionName must be the name of a Macintosh Toolbox function. FB³ recognizes the names of hundreds of Toolbox functions and procedures; advanced programmers can also use the `Toolbox` statement and the `TBAlias` statement to add new Toolbox function/procedure names.

The use of the *modifiers*, *addrExpr&* and *arg* parameters is identical to their use in the `Call <Toolbox>` statement. See the description of the `Call <Toolbox>` statement for more information.

See Also:

`Call <Toolbox>`

Fn <userFunction>**statement/function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
[result =] Fn functionName[(param1 [, param2 ...])]
```

Description:

Executes the user function specified by *functionName*, and optionally returns a numeric or string result. The user function must be one which was defined or prototyped at an earlier location in the program. A user function is defined using `Local Fn`, `Long Fn` or `Def Fn <expr>` statement. A user function is prototyped using the `Def Fn <proToType>` statement.

If the user function returns a value, you can use `Fn <userFunction>` as part of a numeric or string expression, as in this example:

```
count% = 3 * Fn NumFish%(x) + 7
```

If the user function does not return a value, then you should use `Fn <userFunction>` as a standalone statement.

If the function definition includes a list of parameters, then you must provide the same number of parameters (in *param1*, *param2*, etc.) when you call the function, and the parameters that you pass must be of compatible types. The compatible types are summarized here (not all of these are available for all kinds of functions; see the individual descriptions of `Local Fn`, `Long Fn` and `Def Fn <expr>`):

<i>Formal variable type (in FN definition)</i>	<i>Compatible types (in FN call)</i>
signed/unsigned byte (<code>var`</code> ; <code>var``</code>)	Any numeric expression ^{1,2}
signed/unsigned short integer (<code>var%;</code> <code>var%`</code>)	Any numeric expression ^{1,2}
signed/unsigned long integer (<code>var&;</code> <code>var&`</code>)	Any numeric expression ^{1,2}
pointer variable (<code>p As Pointer [To unType]</code>)	A record variable, or a long-integer expression ⁶
single/double precision floating point (<code>var!;</code> <code>var#</code>)	Any numeric expression ²
string variable (<code>var\$</code>)	Any string expression ³
address reference (<code>@adr&;</code> <code>@p As Pointer [To unType]</code>)	Any variable (of any type), or a long-integer expression preceded by “=” ⁴
array declaration (<code>tableau[suffixe] (dim1[, dim2...])</code>)	Base array element (<code>arr[suffix] (0[, 0...])</code>) ⁵

Notes:

1. Non-integer values are rounded to integers before being moved into integer or pointer variables.
2. If you pass a numeric value that's outside the range of the formal variable type, you may get an unexpected result, or you may get an overflow error.
3. If you pass a string value that is longer than the maximum size of the formal variable, the string will be truncated.
4. If you specify a variable here, the variable's address will be copied into the formal parameter (`addr&` or `p`). If you specify a long-integer expression preceded by "=", then the value of that expression is copied into `addr&` or `p`.
5. The array must be a numeric or string array (not an array of records). All of the array's elements are accessible to the function. Any changes that are made to the array's elements within the function will also affect the array outside of the function. Note that if the array specified in the formal `Fn` definition has a different type or different dimensions from the array you pass when you call the function, you may get unexpected results, or even a crash. (Be sure you know what you're doing before you try this!)
6. If you specify a record variable here, the record's address will be copied into the formal parameter (`p`). If you specify a long-integer expression, then the value of that expression is copied into `p`.

If the function definition does not have a list of parameters, then you must not include any parameters (nor parentheses) when you call the function.

In most cases, the parameters that you specify in `Fn <userFunction>` are "passed by value." That means that the user function receives a private copy of the parameter's value; if the function changes that copy, it doesn't affect the value of the parameter that was used in the `Fn` call.

In a few cases, the parameters that you specify in `Fn <userFunction>` are "passed by reference." That means that the user function receives the address of the parameter that you specified. If the function changes the contents at that address, it will affect the value of the parameter you passed. Parameters are passed by reference when you use the following kinds of formal parameter declarations in the function definition:

- pointer (`p As Pointer [To someType]`). (Parameter is passed by reference if you specify a record variable when you call the function.)
- address reference (`@addr& p As Pointer [To someType]`). (Parameter is passed by reference if you specify a variable when you call the function.)
- array declaration (`arr[suffix] (dim1[, dim2...])`)

You can also pass the address of a variable or array by using the `VarPtr` function (`VarPtr(var)` or `@var`) when you call the function (if you do this, then specify a long integer variable or a pointer variable as the formal parameter in the `Fn` definition). This is another way to give the function direct access to the memory comprising the variable or array, allowing it possibly to change its value.

Note that there is no way to pass the contents of a record directly to a function. To give the function access to a record's contents, either pass the record by reference, or pass the record's address directly (passing `VarPtr (recVar)` or `@recVar` when you call the function).

A `Fn <userFunction>` call may appear anywhere below the place where the function is defined (or prototyped). It can appear in the “main” scope of the program, or inside other functions. It may even appear inside the very function that it is calling--this allows you to implement so-called “recursive” functions (functions which call themselves).

See Also:

`Local Fn; Long Fn; Def Fn`

Folder**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

folderRefNum%      = Folder(path$, refNum%)
parentFolderVRef% = Folder(":", refNum%)

```

Revision:

June 14, 2000 (Release 3)

Description:

Depending on the parameters used, this function does several things with folders (directories). It can:

- Return a reference number for the current default directory.
- Return a reference number for a directory that you specify.
- Change the current default directory.
- Create a new folder.
- Return the parent volume

When `Folder` returns a reference number, it will either be a volume reference number (if the directory is a root directory), or a working directory reference number (if the directory is not a root directory). You can use this reference number to identify the directory when you use other FB^3 statements such as `Open`.

FUTUREBASIC REFERENCE

This table summarizes the effects of the various parameter combinations. Note that when you use a nonzero number in *refNum%*, you can specify either a working directory reference number, a volume reference number, or a drive ID number. When you specify a volume reference number or a drive ID number, it represents the root directory on the specified volume.

<i>path\$</i>	<i>numRef%</i>	<i>Result</i>
blank	zero	A reference number for the current default directory is returned in <i>folderRefNum%</i>
partial path name	zero	The path is assumed to be relative to the current default directory. If it specifies an existing folder, then a reference number for the folder is returned in <i>folderRefNum%</i> . Otherwise, <i>folderRefNum%</i> returns zero.
full path name	zero	If the path specifies an existing folder, then a reference number for the folder is returned in <i>folderRefNum%</i> . Otherwise, <i>folderRefNum%</i> returns zero.
blank	nonzero reference numbe	The directory indicated by <i>refNum%</i> becomes the current default directory. The value of <i>refNum%</i> is returned in <i>folderRefNum%</i> .
partial path name	nonzero reference number	The path name is assumed to be relative to the directory indicated in <i>refNum%</i> . If the item specified in the path doesn't exist, a new folder with that name is created, and its reference number is returned in <i>folderRefNum%</i> . If the path specifies an existing folder, a reference number for the folder is returned in <i>folderRefNum%</i> . If the path is invalid or it specifies an existing file, then <i>folderRefNum%</i> returns zero.
full path name	any nonzero number	If the item specified in the path doesn't exist, a new folder with that name is created, and its reference number is returned in <i>folderRefNum%</i> . If the path specifies an existing folder, a reference number for the folder is returned in <i>folderRefNum%</i> . If the path is invalid or it specifies an existing file, then <i>folderRefNum%</i> returns zero.
":"	any nonzero number	If the path name is a colon, this return the reference number of the parent folder. If there is no parent folder, then it return the same number that was sent as the <i>refNum%</i>

Note:

The reference number returned in *folderRefNum%* is a temporary number, which is only valid until your program quits. You cannot use this same number to refer to this folder at a later date. If you need to keep track of a file's location over time, create an alias record for the file.

See Also:

System function; Close Folder; Files\$ <index>

For statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
For indexVar = firstValue To lastValue [Step StepValue]
    [statementBlock]
Next [indexVar]
```

Description:

The `For` statement marks the beginning of a “for-loop,” which must end with a `Next` statement. A for-loop is useful when you want to repeat the execution of a block of statements for a particular number of times. This is what happens when a for-loop is encountered:

1. The value of *firstValue* is assigned to *indexVar* (*indexVar* must be a simple numeric variable).
2. The statements in *statementBlock* are executed. *statementBlock* can contain any number of statements, possibly including other for-loops (but note that any for-loop that’s “nested” within *statementBlock* should not use the same *indexVar* as the “outer” for-loop).
3. The value of *indexVar* + *StepValue* is assigned to *indexVar*. (If you omit the `Step StepValue` clause, then incremental value defaults to 1.)
4. The new value of *indexVar* is compared with *lastValue*, to see whether the loop should be repeated:
 - If *StepValue* is positive, then repeat the loop (go to Step 2) if *indexVar* <= *lastValue*.
 - If *StepValue* is negative, then repeat the loop (go to Step 2) if *indexVar* >= *lastValue*.

Notice that the statements in *statementBlock* are always executed at least once.

Normally, you would use a positive value in *StepValue* (or the default value of 1) when *firstValue* is less than *lastValue*. This allows the *indexVar* to “count up” from *firstValue* to *lastValue*. Likewise, you would normally use a negative value in *StepValue* when *firstValue* is greater than *lastValue*; this allows the *indexVar* to “count down” from *firstValue* to *lastValue*. You should not set *StepValue* to zero; this would essentially cause the loop to be repeated forever.

The expressions in *lastValue* and in *StepValue* are re-evaluated after each iteration of the loop. This has a couple of consequences:

- You can, if you want to, dynamically re-assign the values of *lastValue* and/or *StepValue* while the loop is executing (by means of statements within *statementBlock*). This can be useful sometimes.
- If you don’t intend to dynamically re-assign these values, then you should use constants or pre-calculated values for *lastValue* and *StepValue* to increase execution speed.

For example, consider this loop:

```
For n = 3 To Sqr(x!) Step 2
:
Next
```

In the above, the `Sqr` function is called after each iteration of the loop. Assuming that the value of `x!` doesn't change within the loop, we are needlessly recalculating the same `Sqr` value at each iteration. It would be much faster to do it this way:

```
sqrX! = Sqr(x!)
For n = 3 To sqrX! Step 2
:
Next
```

Here the `Sqr` function is called only once.

Example:

Sometimes it's useful to exit a for-loop "early," after some condition within *statementBlock* has been met. The standard way to do this is to use `Exit For`.

```
For p = 1 To maxStrings
  Long If strArray$(p) = searchString$
    found = _zTrue
    theIndex = p
    p = maxStrings 'Force early exit from loop
  End If
Next
```

Note:

The `While` and `Do` statements provide other useful kinds of loop structures.

See Also:

`While`; `Do`, `Exit For`

Frac

function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

fractionValue# = **Frac**(*expr*)

Description:

This function returns the fractional portion of the floating-point expression given by *expr*. If *expr* is negative, then the value returned by `Frac` will also be negative.

Example:

```
Print Frac(78245.1096)
```

Program output:

0.1096

See Also:

`Fix`; `Int`

Get Field

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
Get Field ZTXTHandle&, efID
```

Description:

Creates a “ZTXT”-formatted block which duplicates the text and style contents of the edit field specified by *efID* in the current output window. A handle to the block is returned in *ZTXTHandle&*, which must be a long-integer variable or a *Handle* variable. You can save this information to disk (using, for example, the `Write Field` statement), or you can copy it to another edit field, using the `Edit Field` statement or the `Edit$` statement. A “ZTXT” block is formatted as follows:

<i>Byte offset</i>	<i>Field</i>	<i>Description</i>
0	numChars%	A short (2-byte) integer indicating how many text characters follow. It counts only the text characters, not the style information.
2	text	“numChars%” bytes of text (up to 32,767 bytes).
numChars% + 2	style info (optional)	The style information, if any. You can determine the number of bytes of style information by subtracting (numChars%+2) from the total size of the ZTXT block.

Note:

Your program should dispose of *ZTXTHandle&* (using, for example, the `Def DisposeH` or `Kill Field` statement) when you’re finished using the ZTXT block that the handle refers to.

See Also:

`Edit Field`; `Edit$ statement`; `Def DisposeH`; `Kill Field`; `Write Field`

Get Preferences

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Get Preferences prefFileName$, prefRecord
```

Revision:

February, 2002 (Release 6)

Description:

This statement locates the named preference file and loads the contents of the file into the indicated preferences record. If the size of the record and the size of the file are different, the smaller of the two sizes will be used.

A full example of using the new Preferences commands can be found on page 453.

See Also:

```
Put Preferences; Kill Preferences; Menu Preferences
```


Get Window

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
Get Window [#]wndID, WindowPtrVar&
```

Description:

This statement returns a pointer to the window record for the window specified by *wndID*. The pointer is returned into *WindowPtrVar&*, which must be a long-integer variable or a *Pointer* variable. A window record is a data structure which contains information about the window, such as its current visibility status, current background & foreground colors, and much, much more. In addition, you need to specify a window record pointer whenever you call any Toolbox routine that deals with windows. For information about the contents of the window record, see the “Window Manager” chapter of *Inside Macintosh: Macintosh Toolbox Essentials*, as well as the descriptions of *grafPort* and *CGrafPort* data structures in *Inside Macintosh: Imaging with QuickDraw*.

FB³ uses the window record’s *RefCon* field to maintain information about the window, including its class, its modality (modal or not), its ID number, and its type. This information is only valid for windows created using the *Window* statement. You can access this information as follows:

```
refCon& = WindowPtr&.wRefCon&
class    = (refCon& And &FF)
isModal  = ((refCon& And &FF00) <> 0)
WindowID = (refCon& And &FF0000) >> 16
WindowType = ((refCon& And &FF000000) >> 24) + 1
```

Note:

Your program should not dispose of the window record (i.e., don’t pass the pointer to *Fn DisposePtr*). The system automatically disposes of it when you close the window.

See Also:

Window statement; *Window* function; *Usr WPtr2WNum*

GetProcessInfo

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
GetProcessInfo index%, processName$ [, PSN]
```

Revision:

May, 2001 (Release 5)

Description:

A "Process" is something that is currently running on your computer; this includes, but is not limited to things like applications, control strip extensions, and background applications.

The index parameter in this call indicates which process is to be queried. An index value of -1 means that the front process is used. This is generally the running FB application that you created.

Index values of zero or higher represent running processes. You may climb this list, examining processes as you go, until the process name comes back as a null string. At that point, you have exhausted the system's list of processes and you can quit searching.

The process serial number is an 8 byte value (2 long integers) that holds a unique value which cannot be used by any other concurrent process. You create a process serial number as follows:

```
Dim psn As ProcessSerialNumber
```

Example:

The following example shows how to display a list of running processes.

```
Dim indx&
Dim ProcessName$
Dim psn As ProcessSerialNumber

GetProcessInfo -1, ProcessName$
Print "My name is: "; ProcessName$; ""
Def Tab 10
Print "  indx", "0x-----PSN----- ", "Process Name"
indx& = 0
Do
  GetProcessInfo indx&, ProcessName$, psn
  Long If ProcessName$[0]
    Print indx&, "0x"; Hex$(psn.highLongOfPSN);
    Print Hex$(psn.lowLongOfPSN), ProcessName$
  End If
  Inc(indx&)
Until Len(ProcessName$) == 0
```

See Also:

On AppleEvent, AppleEventMessage\$, Kill AppleEvent, SendAppleEvent

Globals

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Globals "filename1" [, "filename2"...]
```

Description:

This statement behaves identically as the `Include` statement. The keyword `Globals` is maintained for backwards compatibility with earlier versions of FutureBASIC. To make your program easier to read, you may typically use the `Globals` statement to include files which define global variables, constants, record structures, etc., while using the `Include` statement to include functions, etc. However, the `Globals` statement and the `Include` statement are completely interchangeable.

See Also:

```
Include
```

Gosub

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Gosub {lineNumber|"statementLabel"}
```

Description:

Executes the subroutine located at the indicated line number or statement label. The subroutine should include a `Return` statement; `Return` causes execution to continue at the statement following the `Gosub` statement.

`Gosub` is an outdated method for executing routines; it's generally a better idea to encapsulate your routines in a `Local Fn` function. However, there are a couple of possible advantages to using `Gosub`:

- Routines called using `Gosub` may execute somewhat faster than local functions.
- You can create a “private” subroutine inside a local function, and use a `Gosub` within that local function to call the subroutine. The variables used in the subroutine will have the same scope as the local function. This may be a good way to execute certain repetitive tasks within the local function.

Example:

Subroutines can be executed in a “nested” fashion; i.e., one subroutine may call another. FB^3 keeps track of where each `Return` statement should “return” to.

```
Print "First line."
Gosub "sub1"
Print "Fifth line."
End

"sub1"
Print "Second line."
Gosub "sub2"
Print "Fourth line."
Return

"sub2"
Print "Third line."
Return
```

Program output:

```
First line.
Second line.
Third line.
Fourth line.
Fifth line.
```

Note:

It is possible for a `Gosub` statement inside a local function to jump to a subroutine located outside of that function; it's also possible for a `Gosub` statement in "main" to jump to a subroutine located inside a local function. However, this kind of programming is not recommended, because the resulting switches in variable scope can produce unexpected results.

See Also:

```
Return; Fn <userFunction>; Local Fn
```

Goto statement

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
Goto {lineNumber|"statementLabel"}
```

Description:

Causes program execution to continue at the statement at the indicated line number or statement label. The target statement must be within the same “scope” as the `Goto` statement (i.e., they must both be within the “main” part of the program, or they must both be within the same local function). Also, you should never use `Goto` to jump into the middle of any “block” statement structures (such as `For...Next`, `Select...End Select`, `Long If...End If`, etc.).

`Goto` is sometimes useful in the “main” part of the program, to branch around certain structures such as `EnterProc` procedures. However, excessive use of `Goto` can lead to code that is difficult to read and maintain. The use of `Goto` is generally discouraged; FutureBASIC’s other branching and looping structures offer a better solution.

See Also:

`Local Fn`; `Gosub`; `For`; `While`; `Do`; `Long If`; `Select`; `EnterProc`

HandleEvents

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

HandleEvents

Description:

`HandleEvents` performs a number of important functions that affect the user's experience. It examines the system event queue, as well as FutureBASIC's internal event queues, to see whether any recent events have occurred for your program, that have not yet been handled. If any such events are found, `HandleEvents` removes them from the queue and responds to them. `HandleEvents` also performs the important function of turning over control to the Process Manager. The Process Manager oversees the execution of all processes on your Macintosh; once it has control, the Process Manager may allow another application to run for a short time before returning control to your application.

`HandleEvents` responds to some kinds of user actions by calling functions that you have designated in your program. It responds to other kinds of user actions in predetermined, "automatic" ways.

"Automatic" responses by HandleEvents

- Allows the Process Manager to bring another process to the front, if the user has selected it in the Applications menu or clicked in one of its windows.
- Opens menus and tracks selection, if user has clicked on the menu bar.
- Activates an inactive window, if the user has clicked on the window's structure region (e.g. its title bar). (This action is inhibited if the window's `_keepInBack` attribute is set.)
- Handles dragging & resizing of the active window.
- Performs "standard" handling of mouseclicks and keystrokes in the currently active edit field (if any).
- Highlights & tracks various objects when they're clicked (e.g. buttons, window close box, etc.)
- For any window that requires updating: redraws all buttons, scrollbars, edit fields and picture fields (unless the window's `_noAutoClip` feature is set). Also redraws certain parts of the window's structure region.
- If the user presses ⌘-period, and no `On Break Fn` function has been identified, then `HandleEvents` displays a dialog asking whether the user wants to stop or to continue. If the user elects to stop, `FB^3` then calls your designated `On Stop Fn` function (if any), and then halts the program.

Note: You can inhibit and/or alter these responses by trapping low-level events (especially the `_mButDwnEvt` event) in a system event-handling function. See below for more details.

“Programmed” responses by HandleEvents

There are many kinds of common user actions, such as button clicks and menu selections, which you will want to explicitly handle with program statements. When you write a function that is to handle events of a certain type, you designate it as an event-handling function by executing statements like `On Dialog Fn <functionName>`, or `On Menu Fn <functionName>`. Once you’ve designated your event-handling function(s) this way, `HandleEvents` will examine recent user actions to determine whether any of them are of the kind that your function(s) can handle. If any such events are found, `HandleEvents` calls the appropriate event-handling function once for each such event. See the descriptions of the various `On <eventType>` statements, to learn what types of user actions can be handled.

If you haven’t identified a function to handle a certain class of user actions, then `HandleEvents` just ignores actions of that class. For example, if you have not identified any function with the `On Dialog` statement, then `HANDLEEVENTS` will ignore button clicks and other similar actions. `HandleEvents` will still perform the “automatic” responses listed above, however.

Intercepting system events

There may be times when you need greater control over how `HandleEvents` responds to certain events. For example, you may want to inhibit or alter some of the “automatic” responses that `HandleEvents` normally performs. To do this, you should designate one of your functions as a “system event-handling function,” by using the `On Event` statement. Once you’ve designated such a function, `HandleEvents` calls that function first, before it executes any of its “automatic” responses and before it calls any of the other event-handling functions you may have designated. `HandleEvents` either passes a system event to your function (if there’s an event in the queue), or it passes a “null event” to your function (if there are no events in the queue).

After your system event-handling function returns, `HandleEvents` continues to handle that same event, unless it was a null event. Depending on what the event was, `HandleEvents` may perform some of its “automatic” responses, or it may call another one of your event-handling functions. If you don’t want `HandleEvents` to continue handling the event after your system event-handling function exits, then you need to “trick” FB³ into thinking that the event was a null event. You do this by executing a line like the following, in your system event-handling function, after you’ve handled the event:

```
theEvent&.evtNum% = _nullEvt
```

where `theEvent&` is a pointer to the event record.

In order to provide the user with snappy response to actions, and to share execution time with other processes, your program should call `HandleEvents` as often as possible. Most well-designed programs contain a “main event loop” which calls `HandleEvents` repeatedly for as long as the program is executing, allowing `HandleEvents` to call the various event-handling functions as events occur.

See Also:

`On <eventType> statements`; `Dialog statement/function`; `Menu function`;
`Mouse function`; `Event function`; `TEKey statement/function`

Handshake

statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:****Handshake** *portID*, *handshakeType***Description:**

Sets the handshaking parameter for the open serial port specified by *portID* (which can be either `_modemPort` or `_printerPort`). The handshaking parameter determines how i/o operations will be negotiated when your program subsequently writes to or reads from the specified port. *handshakeType* can take any of the following values:

<i>Constante</i>	<i>Value</i>	<i>Description</i>
<code>_none</code>	0	No handshaking. This is the default value.
<code>_CTS</code>	1	CTS (Clear To Send) hardware handshaking. When the computer is ready to send a block of data, it sets the CTS signal. The computer must then wait for a DTR signal from the other device, before sending the data.
<code>_DTR</code>	2	Sets the DTR (Data Terminal Ready) signal on.
<code>_DTRToggle</code>	-2	Sets the DTR signal off.
<code>_XON</code>	-1	XON/XOFF software handshaking. Each device sends an XON character when it's ready to receive data, and sends an XOFF character if its buffer fills up.

To determine the best handshaking format for a device, see the device's manual.

Note:

You must open the serial port (use the `Open "C"...` statement) before executing the `Handshake` statement.

Powerbooks require an initialization setting of `_none` to preset the serial port properly.

See Also:

`Open "C"; Lof`

Hex\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
hexString$ = Hex$(expr)
```

Description:

This function returns a string of hexadecimal digits which represent the integer value of *expr*. The returned string will consist of either 2, 4 or 8 characters, depending on which of `DefStr Byte`, `DefStr Word` or `DefStr Long` is currently in effect. Note that if the value of *expr* is too large to fit in a hex string of the currently selected size, the string returned by `Hex$` will not represent the true value of *expr*.

In FB³, integers are stored in standard “2’s-complement” format, and the values returned by `Hex$` reflect this storage scheme. You need to keep this in mind when interpreting the results of `Hex$`, especially when *expr* is a negative number. For example: `Hex$(-3)` returns “FD” when `DefStr Byte` is in effect; “FFFD” when `DefStr Word` is in effect; and “FFFFFFFFFD” when `DefStr Long` is in effect.

Note:

To convert a string of hex digits into an integer, use the following technique:

```
intVar = Val("&H" + hexString$)
```

`intVar` can be a (signed or unsigned) byte variable, short-integer variable or long-integer variable. Byte variables can handle a *hexString\$* up to 2 characters in length; short-integer variables can handle a *hexString\$* up to 4 characters in length; long-integer variables can handle a *hexString\$* up to 8 characters in length.

See Also:

```
Oct$; Bin$; DefStr Byte/Word/Long; Val&
```

If**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
If expr Then {dest1|statement1 [:statement2 ...]} →
      [Else {dest2|statement3 [:statement4 ...]}]
```

dest1 and *dest2* are either line numbers or quoted statement labels.

Description:

Conditionally executes one or more statements, or jumps to an indicated line, based on the value of *expr*. If *expr* is evaluated as “true” or as nonzero, then the program either jumps to the line indicated by *dest1*, or it executes *statement1*, *statement2*, etc. If *expr* is evaluated as “false” or as zero, and you’ve included the **Else** clause, then the program either jumps to the line indicated by *dest2*, or it executes *statement3*, *statement4*, etc. Each of the *statement*'s can be any executable statement except a “block” statement such as **For**, **While**, **Do**, etc.

expr can be either a numeric expression, a “logical” expression, or a string. A logical expression normally contains “data comparison” operators, and can be evaluated as being either “true” or “false.” Here are some examples of logical expressions:

```
x! > 19.7
myName$ = "RICK"
6*7 <= 42
```

In FB^3, numeric expressions and logical expressions are interchangeable. When a numeric expression is used in the context of a logical expression, then it’s considered “true” if it’s nonzero, or “false” if it’s zero. For example:

```
If x+3 Then Beep
```

Here, the **Beep** will be executed if and only if *x+3* is not zero.

When a logical expression is used in the context of a numeric expression, then it’s evaluated as -1 if it’s true, or as 0 if it’s false. For example:

```
found = (fileName$ = seekName$)
```

Here, if *fileName\$* equals *seekName\$*, the value -1 is assigned to *found*; otherwise, *found* is assigned a value of 0 .

FUTUREBASIC REFERENCE

You can use the operators `And`, `Or`, `Not` within `expr`. Note, however, that these three are considered to be arithmetic operators, not logical operators. This can lead to some unexpected results if you're not careful. For example, this expression:

```
firstNumber& And secondNumber&
```

may evaluate to zero (false), even when both `firstNumber&` and `secondNumber&` are each nonzero (true). When you wish to use `And`, `Or`, or `Not` in the context of a logical expression, you should use operands which always evaluate either to `-1` or to `0`. For example:

```
firstNumber& <> 0 And secondNumber& <> 0
```

This expression behaves “logically,” because `(firstNumber& <> 0)` is always `-1` or `0`; and likewise `(secondNumber& <> 0)` is always `-1` or `0`.

The `expr` can also be a string. When a string is used in the context of a logical expression, it's evaluated as “true” if and only if the length of the string is greater than zero.

Note:

The `If` statement is a one-line structure. To create a conditional structure spanning multiple lines, use the `Long If` statement.

Use caution when comparing floating point values to zero or to whole numbers. The following expression may not evaluate as expected:

```
If x# = 1
```

In this statement, the compiler compares the value in `x#` to an integer "1". Since SANE and PPC math both use fractional approximations of numbers, the actual value of `x#`, though very close to one, may actually be something like 0.99999999 and therefore render unexpected results.

See Also:

`Long If`; `And`; `Or`; `Not`; `Select Case`

Inc**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Inc(intVar)
numericVariable += IntegerValToAdd
```

Revision:

May 30, 2000 (Release 3)

Description:

This statement increments *intVar* by 1; that is, it adds 1 to the value in *intVar*, and stores the result back into *intVar*. *intVar* must be a (signed or unsigned) byte variable, short-integer variable or long-integer variable. If *intVar* is already at the maximum value for its variable type, then `Inc(intVar)` will cycle it back to its minimum value.

Example:

```
Inc(x&)
```

and...

```
x& += 1
```

...is equivalent to:

```
x& = x& + 1
```

.. or:

```
x&++
```

The following expressions are also equivalent.

```
x& = x& + 100
x& += 100
```

Note

The `+=` syntax may not be used for arrays of strings, containers, or records (though it may be used in the concatenation of simple strings or containers). Where arrays are involved, only numeric values may take advantage of this syntax. This syntax is valid for values other than 1.

See Also:

```
Dec; Inc Long/Word/Byte; Def Cycle
```

Inc Long/Word/Byte**statements**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Inc {Long|Word|Byte} (addr&)
```

Description:

This statement increments the long integer, short integer or byte which begins at the specified address in memory; that is, it adds 1 to the value in memory and stores the result back into the addressed location. If the long integer, short integer or byte is already at its maximum possible value, then the statement will cycle it back to its minimum value.

Example:

```
Inc Long (myAddr&)
```

...is equivalent to:

```
Poke Long myAddr&, Peek Long(myAddr&) + 1
```

Also:

```
Inc Word (myAddr&)
```

...is equivalent to:

```
Poke Word myAddr&, Peek Word(myAddr&) + 1
```

See Also:

```
Inc; Dec Long/Word/Byte; Def Cycle
```

Include statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Include {"<path>"|alisResID}
```

Description:

When the compiler encounters an `Include` statement, it looks for a text file indicated by `<path>` or `alisResID`, reads the FB^3 statements contained in that file, and effectively inserts those statements into the source code stream. `<path>` can be either a full or partial pathname; if you use a partial pathname, it's assumed to be relative to the folder containing the "parent" source file (i.e., the source file that has the `Include` statement). `<path>` may refer either to a text file, or to an alias file whose target is a text file. `alisResID` must be the resource ID number of an "alis" resource whose target is a text file. This resource should exist in the resource fork of the "parent" source file. Include files may be "nested"; that is, an Include'd file may contain references to other Include files.

Example:

Suppose we create a file "Assign.Incl" which contains the following lines of text:

```
a = 3
b = 7
```

Now suppose we write a program like this:

```
Dim x(100)
Include "Assign.INCL"
c = a + b
```

When we compile this program, the result will be identical to this:

```
Dim x(100)
a = 3
b = 7
c = a + b
```

Note:

FutureBASIC's Project Manager is generally a more convenient way to combine the source code from several different files. However, there are some advantages to using the `Include` statement, such as the ability to conditionally include files (see `Compile Long If`).

FB^3 does not allow a given file to be included more than once in the source stream. If a second `Include` reference is made to a given file, that `Include` statement will be ignored.

See Also:

`Compile Long If`

Index\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
stringVar$ = Index$(element [,indexID])
```

Description:

This function returns an element from one of the special `Index$` string arrays (see the `Index$` statement and the `Clear <index>` statement for information about how `Index$` arrays are created). The *indexID* parameter specifies which of the `Index$` arrays (0 through 9) to return an element from; if this parameter is omitted, `Index$` array 0 is used. The *element* parameter specifies which element to get. If the indicated `Index$` array has not yet been initialized by the `Clear <index>` statement, or if *element* specifies an element that has not yet been assigned any value, then `Index$` returns an empty (zero-length) string.

See Also:

```
Index$ statement; IndexF; Index$ I; Index$ D; Clear <index>
```


Index\$

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Index$(element [,indexID]) = stringExpr$
```

Description:

This statement assigns the string specified by *stringExpr\$* to an element in one of the special Index\$ string arrays. The Index\$ string arrays have some features which are not available in a “normal” string array:

- Under the right circumstances, they can occupy less memory than a “normal” string array;
- There are certain useful operations that can only be performed on Index\$ string arrays (see the Index\$ D, Index\$ I statements, and the IndexF function).

The Index\$ string arrays are always global in scope. You can read the contents of an Index\$ array element by using the Index\$ function.

Your program can use up to ten Index\$ arrays, numbered 0 through 9. The *indexID* parameter specifies which Index\$ array to use; if you omit this parameter, Index\$ array 0 is used. The *element* parameter specifies which element to set within the selected Index\$ array.

Before you can assign values to an Index\$ array, your program must allocate space for the desired Index\$ array by executing one of the variants of the `Clear <index>` statement (either “Syntax (1)” or “Syntax (2)”). You must execute a separate `Clear <index>` statement for each of the Index\$ arrays (up to 10) that you wish to use. Depending on which variant of `Clear <index>` you used to initialize the array, there may be additional restrictions on how you use the Index\$ statement:

- If you used the `Clear numElements&,indexID,eltSize` variant, then the *element* parameter you specify in the Index\$ statement should not exceed `numElements& - 1`, and the length of the string you assign should not exceed `eltSize - 1` characters.
- If you used the `Clear bytes& [,indexID]` variant, then the total memory occupied by all the strings you assign to that Index\$ array must not exceed `bytes&`.

You can use the `Mem` function to determine whether you’re approaching these limits, and you can use the `Clear <index>` statement to increase an array’s memory allocation if necessary.

Memory allocation

If the strings in your `Index$` array are to be of varying lengths, then you can save memory by specifying variable-length cells when you allocate memory for the array (use the `Clear bytes& [,indexID]` syntax to do this). This is because of the two different storage schemes used by variable-length cells vs. fixed-length cells:

- Fixed-length cells (`Clear numElements&,indexID,eltSize`) always use `eltSize` bytes for every string in the array: this means there may be “unused” bytes following any giving string:

```
Clear numElements&, indexID, 10
```

05	B	o	w	i	e				
06	T	u	c	s	o	n			
08	S	u	n		C	i	t	y	
08	P	r	e	s	c	o	t	t	
06	A	p	a	c	h	e			

Actual memory needed for these 5 elements: 50 bytes

- Variable-length cells (`Clear bytes& [,indexID]`) are stored contiguously in memory: there are no “wasted” bytes between strings:

```
Clear numBytes&, indexID
```

05	B	o	w	i	e				
06	T	u	c	s	o	n			
08	S	u	n		C	i	t	y	
08	P	r	e	s	c	o	t	t	
06	A	p	a	c	h	e			

Actual memory needed for these 5 elements: 38 bytes

The tradeoff is this: although variable-length cells can save you memory, they also involve more complex internal memory management. Operations involving fixed-length cells generally execute somewhat faster.

Note:

You can use the `Mem` function to retrieve various kinds of information about the status of an `Index$` array.

See Also:

```
Index$ function; IndexF; Index$ I; Index$ D; Clear <index>; Mem
```

Index\$ D

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

`Index$ D (element [, indexID])`

Description:

This statement deletes an element from one of the special `Index$` string arrays (see the `Index$` statement and the `Clear <index>` statement for information about how `Index$` arrays are created). The `indexID` parameter specifies which of the `Index$` arrays (0 through 9) to delete an element from ; if this parameter is omitted, `Index$` array 0 is used. The `element` parameter specifies which element to delete.

The `Index$ D` statement does not merely assign a null string to the indicated element. Instead, it moves all of the subsequent array elements in memory, in order to fill the “gap” left by the deleted element. As shown in the diagram, this also affect the element numbers by which the moved strings are identified.

This statement is the complement of `Index$ I`, which inserts an element into the array.

Example:

This illustrates the difference between deleting an element using `Index$ D`, and “clearing” an element by assigning a null string to it.

INDEX\$ D(5)			
<i>element #</i>	<i>contents</i>	<i>element #</i>	<i>contents</i>
4	Sandy	4	Sandy
5	Joshua	5	Carrie
6	Carrie		
Before		After	

INDEX\$(5)=""			
<i>element #</i>	<i>contents</i>	<i>element #</i>	<i>contents</i>
4	Sandy	4	Sandy
5	Joshua	5	
6	Carrie	6	Carrie
Before		After	

See Also:

`Index$ statement`; `IndexF`; `Index$ I`; `Clear <index>`; `Mem`

Index\$ I

statement

✓ Appearance	✓ Standard	✓ Console
--------------	------------	-----------

Syntax:

`Index$ I (element [, indexID]) = stringExpr$`

Description:

This statement inserts a new element into one of the special `Index$` string arrays, and assigns the value of `stringExpr$` to the new element (see the `Index$` statement and the `Clear <index>` statement for information about how `Index$` arrays are created). The `indexID` parameter specifies which of the `Index$` arrays (0 through 9) to insert an element into; if this parameter is omitted, `Index$` array 0 is used. The `element` parameter specifies where in the array to insert the string.

Unlike the `Index$` statement, the `Index$ I` statement does not merely replace the contents of the indicated element. Instead, all of the strings at position `element` and beyond are moved in memory, to open a space in which to insert the new string. As shown in the diagram, this also affect the element numbers by which the moved strings are identified.

This statement is the complement of `Index$ D`, which deletes an element from the array.

Example:

This illustrates the difference between inserting an element using `Index$ I`, and replacing an element using the `Index$` statement.

INDEX\$ I(5)="Riley"			
<i>element #</i>	<i>contents</i>	<i>element #</i>	<i>contents</i>
4	Sandy	4	Sandy
5	Joshua	5	Riley
6	Carrie	6	Joshua
		7	Carrie
Before		After	

INDEX\$(5)="Riley"			
<i>element #</i>	<i>contents</i>	<i>element #</i>	<i>contents</i>
4	Sandy	4	Sandy
5	Joshua	5	Riley
6	Carrie	6	Carrie
Before		After	

See Also:

`Index$ statement`; `IndexF`; `Index$ D`; `Clear <index>`; `Mem`

IndexF

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
foundElement = IndexF(string$ [, startElement [, indexID]])
```

Description:

This function searches the `Index$` array specified by `indexID`, for a string which contains the characters specified by `string$`. The search begins at the element specified by `startElement`. If you omit the `startElement` parameter, the search begins at element zero (i.e., at the beginning of the array). If you omit the `indexID` parameter, `Index$` array #0 is searched.

The search is case-sensitive. If a match is found, `IndexF` returns the element number of the matching element; otherwise, it returns `-1`. If you specify an `Index$` array which does not exist (because no space for it has been allocated using the `Clear <index>` statement), `IndexF` returns `-1`.

Example:

```
elementNumber = IndexF("Modem")
```

This code would find a match with the following `Index$` elements:

```
"Modem"
"Modemizing"
"my Modem"
```

The code would not find a match with these elements:

```
"modem"
"MODEM"
"horse feathers"
```

See Also:

```
Index$ statement/function; Index$ I; Index$ D; Clear <index>
```

InKey\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
stringVar$ = InKey$
```

Description:

This function tests whether there is a keypress event pending in the event queue (this happens if the user has pressed a key and no program statement has yet detected it). If there is such an event on the queue, `InKey$` removes the event from the queue and returns a 1-character string indicating what key was pressed. If there is no keypress event pending, `InKey$` returns a null (zero-length) string.

Note that `InKey$` is a rather old-fashioned way to check for keypresses. It will not work reliably if your program calls `HandleEvents` regularly, because `HandleEvents` also checks for keypress events and removes them from the event queue. If your program uses `HandleEvents`, then you should check for keypresses either by trapping the `_evKey Dialog` event (if there are no active editable text fields in the current window or if you are running in the Appearance Compliant runtime), or by checking the `TEKey$` function within your designated `On Edit` routine (if there is an active editable text field in the current window).

Note:

The pressed key character is not automatically displayed on the screen. Use the `Print` statement if you want to display the character.

The value returned by `InKey$` is affected by whether the Shift key, Option key, etc. were down. However, these “modifier” keys will not generate any keypress event by themselves; they must be pressed in combination with some character key in order for `InKey$` to detect the keypress. If you want to detect whether a certain modifier key was down when an event happened, use the `Event%` function (or the `_evtMeta` field of the `Event` record). If you want to detect whether a modifier key is currently down, use the Toolbox function `GetKeys`.

See Also:

```
HandleEvents; Dialog function; TEKey$ function; Event% function;
Event function; On Edit Fn; On Dialog Fn
```


InKey\$ <ioChannel>**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
stringVar$ = InKey$(deviceID)
```

Description:

This function reads a single character from an open serial port or an open file, and returns it as a 1-character string. *deviceID* should equal either `_modemPort` or `_printerPort` (see the `Open "C"` statement), or should be the *fileID* number of an open file (see the `Open` statement).

The function returns an empty (zero-length) string in the following situations:

- There are no characters currently in the input buffer of the specified serial port; or:
- The end of the specified file has been reached.

Note:

This function is similar to the `Read# deviceID, stringVar$;1` statement. However, the `Read#` statement generates an error if you attempt to read past the end of a file.

See Also:

```
InKey$; Read#; Open; Open "C"
```

Input

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Input [@(col,row) | %(h,v)] ["prompt";] var1 [, var2...]
```

Description:

This statement displays an optional prompt in the current output window, then waits for the user to enter data from the keyboard. The entered data is displayed in the window (to the right of the prompt, if any) as the user types it. When the user finishes entering the data, the data are assigned to the variables *var1*, *var2* etc., and the program then continues execution at the next statement. This is a simple (but old-fashioned) way to let the user interact with the program.

Parameters

<i>Paramètre</i>	<i>Description</i>
<i>@(col,row)</i> <i>%(h,v)</i>	This specifies where the input cursor (or the prompt, if it's specified) should appear within the window. If you use the <i>@(col,row)</i> variant, then <i>col</i> and <i>row</i> represent the text column and row where the prompt or cursor should appear; the exact pixel location depends on the current font ID and font size. If you use the <i>%(h,v)</i> variant, then <i>h</i> and <i>v</i> represent horizontal and vertical pixel positions; the prompt (or cursor) is placed with its lower-left corner at point (<i>h,v</i>). If you omit this parameter, then the prompt or cursor is placed at the window's current pen position.
<i>"prompt";</i>	If you specify this parameter, the literal string inside the quotes is displayed as a prompt. The input cursor is displayed just to the right of the prompt.
<i>var1[,var2...]</i>	These must be string or numeric variables (not record variables). The data that the user enters are assigned to these variables according to the rules explained below.

Variable Assignment

The **Input** statement expects the user to enter data as a sequence of data items separated by commas or tabs as in this example:

```
23, green, -15.7
23 [tab] green [tab] -15.7 [cr]
```

Subsequent text will refer to the more common comma delimiter, but works also with tabs.

Each one of the comma-delimited items is assigned (after some conversion, if necessary) to a separate variable in the `var1 [,var2...]` list. If the user enters more items than the number of variables in the list, the extra entered items are ignored. If the user enters fewer items than the number of variables in the list, then zeros or null strings are assigned to the extra variables.

The items may undergo some conversion before being assigned to the variables:

- If the variable is a string variable, leading spaces are stripped from the item (trailing spaces are not stripped).
- If the variable is a numeric variable but the entered item is a string that can't be interpreted as a number, then 0 is assigned to the variable.
- If the variable is a numeric variable and the entered item is numeric, then it is subject to the same kinds of conversion as if the item were assigned via an assignment statement. For example, the item's value may be rounded to an integer, or excess digits of precision may be dropped.

Quoted Items

By surrounding an entered item with double quotes, the user can exercise more control over how the item is assigned to a (string) variable. FB³ always strips the surrounding quotes from the item before assigning the string to the variable; however:

- Leading and trailing spaces inside the quotes are preserved;
- A comma that occurs inside the quotes is considered part of the item, rather than a delimiter between items.

Therefore, if the user needs to input an item which has leading blanks, or which contains a comma, then he or she should surround the item with double quotes when typing it in. To assign an arbitrary line of entered text to a string variable without the need for enclosing quotes, use the `Line Input` statement.

Edit Field vs. Input

`Input` is an old-fashioned way of getting keyboard input. A more appropriate and “Macintosh-like” way to get input is to use the `Edit Field` statement.

Console Behavior:

When you use the Console runtime, the user interface for `Input` is somewhat different:

- The text-position parameters `[@(col,row) | %(h,v)]` are ignored.
- A “Prompt Window” is opened. It displays the prompt string (if any). It contains an edit field into which the user can enter the response, and contains an “OK” button which the user can click to indicate that the response is complete.
- When the user clicks “OK”, the Prompt Window disappears; then FB switches to the Text Window and prints both the prompt and the user's response.

See Also:

`Input#`; `Line Input`; `Edit Field`; `Edit$` function

Input# statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Input# deviceID, var1 [, var2 ...]
```

Revision:

May 30, 2000 (Release 3)

Description:

This statement reads text data from the open file or serial port specified by *deviceID*, and stores the data into the specified variables. The variables *var1*, *var2* etc. must be numeric, container, or string variables (not record variables).

If *deviceID* equals zero, then `Input#` reads data from the keyboard. `Input#0, var1 [,var2...]` is identical to `Input var1 [,var2 ...]`.

The data in the file (or coming in through the serial port) should be formatted as text items delimited by commas and/or carriage-returns. Each item is assigned to a separate variable. Some data conversion may occur during the assignment; see the `Input` statement for more details.

If *deviceID* specifies a file, then `Input#` reads a line of text from the file, beginning at the current “file mark” position (which is usually at the beginning of the line), and ending when a carriage-return character is encountered, or the end of the file is encountered, or 255 characters have been read, whichever occurs first. The file mark is then advanced to a position just past the last character read.

`Input#` then attempts to assign each of the comma-delimited items in the text line to one of the variables (*var1*, *var2* etc.) in the variable list. If there are more items in the text line than variables in the list, the extra items are discarded. If there are fewer items in the text line than variables in the list, then zeros or null strings are assigned to the extra variables.

If *deviceID* specifies an open serial port (i.e., if its value is `_modemPort` or `_printerPort`), then `Input#` behaves in a similar way, except that the concepts of “file mark” and “end of file” generally don’t apply.

`Input#` is the slowest method of reading data from disk. For greater speed, try using `Read#` instead. (Note however that `Input#` and `Read#` generally interpret the data in the file differently.)

`Input#` can read the data created by `Print#`. Note, however, that if you want to create data items that are delimited by commas, you must explicitly print comma characters between them. `Print#` does not automatically insert commas between the items it puts out.

Note:

If the file mark is already at the end of file when you execute `Input#`, FB^3 generates an “Input past end of file” error. To prevent this situation, check the value of `Eof(deviceID)` before executing `Input#`.

See Also:

`Input`; `Line Input#`; `Open`; `Read#`; `Write#`; `Print#`; `Eof`

InStr**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

foundPosition = InStr(startPos, ↵
                      targetString$ | targetContainer$$, ↵
                      searchString$ | searchContainer$$)

```

Revision:

May 30, 2000 (Release 3)

Description:

This function searches for the first occurrence of *searchString\$* or *searchContainer\$\$* within *targetString\$* or *targetContainer\$\$*, starting at character position *startPos*. (If *startPos* is less than 1, it's treated as 1.) If a match is found, the function returns the character position (1..255) within *targetString\$* or *targetContainer\$\$* where the match begins. If no match is found, the function returns zero. The string search is case-sensitive.

Note:

InStr always returns zero in these cases:

- when *startPos* is greater than `Len(targetString$)`;
- when `Len(searchString$)` is zero.

See Also:

`IndexF`; `Mid$`; `Left$`; `Right$`

Int**function**

✓ Appearance**✓ Standard****✓ Console**

Syntax:

```
nearestInteger& = Int(expr#)
```

Description:

Returns the value of *expr#* rounded to the nearest integer.

Note:

Int returns a “long integer” value, which means that *expr#* should be within the range −2147483648 through +2147483547. To obtain the integer part of numbers which are outside this range, use the **Fix** function. (Note however that **Fix** truncates the fraction part rather than rounding to the nearest integer. In general, **Fix** and **Int** don’t return the same values.)

See Also:

Fix, **Frac**

InvalidRect

function

✓ *Appearance***✓** *Standard***✗** *Console*

Syntax:

```
ignored = Fn InvalidRect(rect)
```

Revision:

February, 2002 (Release 6)

Description:

Before Carbon became a part of the Mac toolbox, we were able to use a toolbox procedure called `InvalidRect` to mark a portion of the current window as an area that needed to be refreshed during the next update. This call will not work in OS-X (or in the Carbon version of OS 9). Our substitute, `Fn InvalidRect`, will work in versions 7 through X without additional coding required on your part.

Kill**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Kill path$ [, refNum% [, dirID&]]
```

Description:

This statement deletes a (closed, unlocked) file or folder from the disk. The *path\$* parameter can be either a full or partial pathname; the *refNum%* parameter can be a working directory reference number, a volume reference number, a drive ID number, or zero. The *dirID&* parameter is a directory ID number of the item's parent directory. If *dirID&* is not specified, then the current `ParentID` value (if nonzero) is used instead. `Kill` resets the `ParentID` value to zero after the item is deleted.

The easiest way to use this statement is to specify the item's name in *path\$*, the volume reference number of its volume in *refNum%*, and the ID number of its parent directory in *dirID&*. For other ways to use these parameters, see Appendix A: *Specifying Files and Folders*.

Note:

You cannot delete a folder if it has any contents, nor if the folder is “open.” A folder is considered “open” if your program has executed any statement which returned a working directory reference number for the folder, and has not subsequently closed it using the `Close Folder` statement.

See Also:

`ParentID`, `Close Folder`

Kill AppleEvent

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Kill AppleEvent eventType&, eventClass&
```

Revision:

May, 2001 (Release 5)

Description:

Once established, an Apple Event remains in effect for the life of your program. This includes event handlers that you create as well as event handlers that are created by the runtime. One example might be the apple event that opens a document. Its event type is `_"core"` and its class is `_"odoc"`. If you wanted to override these established settings you would begin by removing the old version.

```
Rem _coreEventType = _core"  
Rem _kAEOpenDocuments = _odoc"  
  
Kill AppleEvent _kRequiredEventClass,_kAEOpenDocuments
```

Now you are ready to establish a new vector.

```
On AppleEvent(_kRequiredEventClass,_kAEOpenDocuments) -  
    Fn myODocHandler
```

Example:

 CD Example: AppleEvents folder

See Also:

```
SendAppleEvent; HandleEvents; On AppleEvent; GetProcessInfo;  
AppleEventMessage$
```

Kill Dynamic

statement

✓ Appearance

✓ Standard

✓ Console

Syntax:

Kill Dynamic *arrayName*

Revision:

May, 2001 (Release 5)

Description:

This statement releases the memory allocated for use by a dynamic array. The array may still be used after the `Kill Dynamic` statement is executed and will begin to grow once again as information is added.

See Also:

`Compress Dynamic`, `Dynamic`, `Read Dynamic`, `Write Dynamic`

Kill Field statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
Kill Field Handle&
```

Description:

This statement disposes of the specified handle. This means that the memory block referenced by *Handle&* gets released, and the value in *Handle&* is thereafter no longer a valid handle. `Kill Field` is commonly used with handles returned by `Get Field` or `Read Field`, but it can dispose of any kind of handle. However, you should specifically NOT use it to dispose of resources, regions, window controls, and other “standard” kinds of Macintosh objects which are created and managed by the MacOS. Instead, you should use the appropriate Toolbox routine (`ReleaseResource`, `DisposeRgn`, `DisposeControl`, etc.) to dispose of such objects.

`Kill Field` is similar to the Toolbox call `DisposeHandle`, except that it (like the `Def DisposeH` statement) checks for `_nil` handles and sets the *Handle&* variable to zero.

See Also:

```
Def DisposeH; Get Field; Read Field; SysError
```

Kill Picture

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Kill Picture pictureHandle&
```

Description:

This statement calls the Toolbox procedure `KillPicture`, which releases the memory occupied by the picture which is specified by `pictureHandle&`. This should be a handle that your program created using the `Picture On`, `Picture Off` statements (or the Toolbox routines `OpenPicture` and `ClosePicture`), or the `Usr GetPICT` statement. You should not use `Kill Picture` to release a picture handle that was created by other means (for example, a picture resource handle).

Warning: Do not use `Kill Picture` to kill a picture that is currently being displayed in a picture field. You must close the picture field first (using the `Edit Field Close` statement) before calling `Kill Picture`.

Warning: You should make sure that `pictureHandle&` does not equal zero before calling `Kill Picture`. Disposing of a “nil handle” can cause problems on some older systems.

See Also:

`Picture`; `Picture On/Off`; `Picture Field`; `Edit Field Close`

Kill Preferences

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Kill Preferences prefFileName$
```

Revision:

February, 2002 (Release 6)

Description:

This routine locates and deletes a file with the specified name in the preference folder. If the file does not exist, this statement does nothing.

A full example of using the new `Preferences` commands can be found on page 453.

See Also:

```
Put Preferences; Get Preferences; Menu Preferences
```

Kill Resources

statement

✓ Appearance

✓ Standard

✓ Console

Syntax:

```
Kill Resources "resType", resID%, ["resType", resID%...]
```

Revision:

October 19, 2000 (Release 4)

Description:

This statement will store specified resource types and IDs in an internal list. When one of these resources is encountered by the compiler, it will not be added to the built application.

There are two important reasons for a call like this. The first is that resources normally included in the runtime shells may be considered unnecessary by a programmer may easily be deleted from the finished product. The second is that multiple resource files may be used and unnecessary resources may be eliminated programmatically.

Example:

To prevent a picture resource with an ID of 8001 from being added to the compiled application, the syntax would be:

```
Kill Resources "PICT", 8001
```

See Also:

Resources

Left\$ and Left\$\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
subString$      = Left$(string$, numChars)
subContainer$$ = Left$$ (container$$, numChars)
```

Revision:

May 30, 2000 (Release 3)

Description:

This function returns a substring or subcontainer consisting of the leftmost *numChars* characters of *string\$* or *container\$\$*. If *numChars* is greater than the length of *string\$* or *container\$\$*, then **Left\$** returns the entire string or container. If *numChars* is zero, then **Left\$** returns an empty (zero-length) string.

Note:

You may not use complex expressions that include containers on the right side of the equal sign. Instead of using:

```
c$$ = c$$ + Left$$ (a$$, 10)
```

Use:

```
c$$ += Left$$ (a$$, 10)
```

See Also:

Mid\$ function; Right\$

Len**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
stringLength = Len(string$|container$)
```

Revision:

May 30, 2000 (Release 3)

Description:

This function returns the number of characters contained in *string\$* or *container\$*. In the case of an empty string, zero is returned.

Note:

To determine the maximum number of characters that can be put into a string variable, use:

```
SizeOf(stringVar$) - 1
```

The maximum number of characters allowed in a container is theoretically 2 gigabytes, but is normally limited by available memory.

See Also:

SizeOf

Let**statement (assignment)**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

1. `[Let] var = expr`
2. `[Let] var;length = address&`

Description:

The `Let` statement assigns a value to the variable `var`, replacing whatever value `var` had before. Note that the `Let` keyword is optional.

If you use Syntax 1, the value of `expr` is assigned to `var`:

- If `var` is a numeric variable, then `expr` can be any numeric expression; if `expr` is outside the range or precision that can be stored in `var`, then the expression will be appropriately converted.
- If `var` is a `Pointer` variable, then `expr` can be `_nil` (zero), or another `Pointer` variable of the same type, or any valid address expression.
- If `var` is a `Handle` variable, then `expr` can be `_nil` (zero), or another `Handle` variable of the same type, or any valid address expression whose value is a handle.
- If `var` is a string variable, then `expr` can be any string expression. You should make sure that the length of `expr` does not exceed the maximum string size that will fit into `var`.
- If `var` is a “pseudo” record (declared using `Dim var.constant`), then `expr` must be a record variable declared with the same length as `var`.
- If `var` is a “true” record (declared using `Dim var As recordType`), then `expr` must be a record variable of the same type as `var`.

If you use Syntax 2, then `length` bytes are copied into `var`, from the memory location starting at `address&`. The `length` parameter must be a static integer expression (i.e., it cannot contain any variables). Note that FB^3 does not check whether `length` actually equals the size of `var`. If `length` is too small, an incomplete value will be copied into `var`; if `length` is too big, data will be copied into addresses beyond `var`’s location in memory (this can be dangerous).

See Also:

`Dim`; `Dim Record`; `Begin Record`; `BlockMove`; `Def BlockFill`; `Constant`
 declaration **statement**

Library

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Library "LibraryName"
Library
```

Description:

Libraries are routines that are stored either in the Extensions folder of the System Folder or in the folder with the compiled application. Libraries are accessed through FB³'s standard toolbox assignment mechanism. Libraries may be dragged into the project window and included in the compiled application.

The first syntax: `Library "LibraryName"` tells FB³ to begin using toolbox calls from *LibraryName*. Only one library at a time is ever accessible. When the toolbox statements have been set up using `Toolbox` and `Toolbox Fn`, the second syntax is used to return to the default library.

The `Library` command is a mechanism that establishes links to a precompiled library. It doesn't matter who created the library (Apple's libraries are most common) or what language it was written in. It's just a file full of subroutines. The creator of the library will usually provide a set of function calls that may be used after the library is opened. Each of these is made available to your FB³ application with a `Toolbox` or `Toolbox Fn` command.

Example:



CD Example: PPC SharedLib example

The following example is already part of the FB³ headers and does not need to be entered. It is offered here as an example only.

```
Library "QuickTimeLib"

Toolbox Fn EnterMovies = OSErr `0x7001,0xAAAA
Toolbox ExitMovies `0x7002,0xAAAA
Toolbox Fn GetMoviesError = OSErr `0x7003,0xAAAA

Library
```

Checking For Available Library Commands

After executing a `Library` statement, you may wish to determine if a particular routine is available. The following example shows how that can be accomplished. A source code example for a library written in C is included on your CD. Our simple shared library creates a toolbox call named `SharedBeepNTimes`.

```

Library "Shared Library"
Toolbox SharedBeepNTimes(Word) `0xA9FF
Library

Dim result,err,connID&,MainAddr&, symAddr&
Dim symClass`,symClassFill`,symCount&,msg$,i

Print "*** Getting connection to shared library ***"

err = Fn GetSharedLibrary("Shared Library",-
    _"pwpc",0x0005,connID&, MainAddr&, msg$)

Long If err
    Print "*** No Library entry by that name... ***"
Xelse
    err = Fn CountSymbols(connID&, symCount&)
    Print "*** Checking For Entry. ***"
    Print "found";symCount&," Total."
    For i=0 To symCount&-1
        symClass` = 0
        err = Fn GetIndSymbol(connID&,i,msg$,symAddr&,symClass`)
        Print i,"Name:'";msg$;"'",
        Print "Addr:";Hex$(symAddr&),"Class:";symClass`
    Next
    err = Fn FindSymbol(connID&, "SharedBeepNTimes",-
        symAddr&, symClass`)
    Long If err
        Print "*** No code entry by that name... ***"
    Xelse
        Print "*** Entry found, trying it... ***"
        SharedBeepNTimes(1)    // Call sharedLibrary
        Print "Done..."
    End If
End If

```

See Also:

FBTestForLibrary; "Insure Validity of PPC Toolboxes" in Editor Manual;
 Toolbox functions; TBAlias

Line function

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
lineAddress& = Line "label"
```

Description:

This function looks for the indicated *label* in your program, and returns the memory address of the first program instruction that follows the label. If you want to use `Line` in this way, the label should indicate the beginning of an `EnterProc` procedure.

Note:

To get an entry point address for an `EnterProc` procedure which is to be used as a “callback procedure” for a MacOS Toolbox routine, use the `Proc` function instead.

See Also:

```
Call; EnterProc; @Fn; Proc
```

Line Input

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Line Input [@(col,row) | %(h,v)] ["prompt";] stringVar$
```

Description:

This statement behaves similarly to the `Input` statement, except that the entire line of text entered by the user (including any commas, quotes and leading spaces) is stored into the single variable *stringVar\$*. See the `Input` statement for a description of the parameters that precede *stringVar\$*.

Note:

`Input` and `Line Input` are considered rather “old-fashioned” ways to interact with the user. Programs with good user-interface design usually utilize Edit Fields instead.

See Also:

`Input`; `Line Input#`; `Edit Field`

Line Input# statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Line Input# deviceID, stringVar$
```

Description:

This statement reads a line of text data from the open file or open serial port specified by *deviceID*, and stores the data into the string variable *stringVar\$*.

If *deviceID* equals zero, then `Line Input#` reads data from the keyboard. `Line Input#0, stringVar$` is identical to `Line Input stringVar$`.

If *deviceID* specifies a file, then `Line Input#` reads a line of text from the file, beginning at the current “file mark” position (which is usually at the beginning of the line), and ending when a carriage-return character is encountered, or the end of the file is encountered, or 255 characters have been read, whichever occurs first. `Line Input#` then assigns the entire string of characters to *stringVar\$*. the file mark is then advanced to a position just past the last character read.

If *deviceID* specifies a serial port (i.e., if its value is `_modemPort` or `_printerPort`), then `Line Input#` behaves in a similar way, except that the concepts of “file mark” and “end of file” generally don’t apply.

Note that `Line Input#` is similar to `Input#`, except that special characters such as commas, quotes and leading spaces are not interpreted as data item delimiters, but instead are copied directly into *stringVar\$*.

Note:

If the file mark is already at the end of the file when you execute `Line Input#`, FB^3 generates an “Input past end of file” error. To prevent this situation, check the value of `Eof(deviceID)` before executing `Line Input#`.

See Also:

`Input#`; `Line Input`; `Eof`; `Open`

Loc**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
result = Loc(deviceID)
```

Description:

This function returns one of two things, depending on the value of *deviceID*.

- If *deviceID* is the file ID number of an open file, the function returns the current location of the file mark as an offset from the beginning of the current record. For example, if `Loc(fileID)` returns zero, the file mark is located at the beginning of the record. The file mark indicates where in the file the next input or output operation will occur.
- if *deviceID* specifies an open serial port (i.e., if its value is `_modemPort` or `_printerPort`), then the function indicates the status of the carrier signal. It returns `_zTrue` if the carrier is detected, or `_false` if the carrier is not detected.

Note:

You can use `Loc` along with `Rec` to determine the exact location of the file mark within the file.

See Also:

`Rec`; `Record`; `Open`; `Lof`; `Handshake`; `Open "C"`

Local statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
[Clear] Local [Mode]
```

Description:

This statement is an alternative way to indicate the beginning of the scope of a local function. If used, it must appear somewhere above the `Local Fn` statement. All non-global variables which are declared between the `Local` statement and the `Local Fn` statement have a scope local to the function. Adding the `Clear` and/or `Mode` keywords has the following additional effects:

- The `Clear` keyword causes all of the function's local variables and arrays (except parameter-list variables) to be initialized to zeros, null strings or empty records, each time the function is called. Otherwise, the variables will have unpredictable initial values. You can accomplish the same effect by adding the `Clear` keyword to the `Local Fn` statement.
- The `Mode` keyword prevents the use of global variables within the function. That is, all variables inside the function will be local variables, even those which have the same names as global variables. This is a good practice when you're writing a function that you might wish to use in a number of different projects, because it removes the possibility of the function's local variables being misinterpreted as globals.

Note:

`Dim` is the only kind of statement that you should put between the `Local` statement and the `Local Fn` statement. Executable statements placed between `Local` and `Local Fn` will never be executed.

You cannot declare any of the the variables in the function's parameter list using a `Dim` statement after the `Local` statement.

A compiler preference allows you to fill `Local FNs` with `$A5A5` for debugging. With this item checked, all functions that do not begin with `Clear Local` have every variable filled with this value. It's a great debugging tool.

See Also:

```
Local Fn; End Fn; Dim; Begin/End Globals
```

Local Fn

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
[Clear] Local Fn functionName [(arg1 [,arg2 ...])]
    [statementBlock]
End Fn [= expr]
```

Description:

This statement marks the beginning of an FB^3 local function. The end of the local function is marked by the `End Fn` statement. A local function is a named collection of statements which can be accessed and executed as a unit by referring to the function's name (see the `FN <userFunction>` statement). All variables and arrays referenced in a local function (except those explicitly declared as "global") are local to the function, which means they do not have any influence outside of the function; any identically-named variables which appear outside of the function are actually different variables, and occupy a different place in memory, than the function's local variables. (An exception to this rule occurs when an array is listed as one of the function's formal parameters; see more about this below.) When your program "calls" (executes) a local function, you can pass data into the function by means of its parameter list (also called its argument list), and you can receive a value back from the function by means of its return value. Local functions allow you to encapsulate complex programming tasks; they're a fundamental and extremely powerful programming construct.

In addition to marking the beginning of the function, the `Local Fn` statement also declares the function's name, the data type of its return value (if any), and the number and types of its input parameters (if any). A local function can be placed anywhere in the program, except inside another local function; you should also not place a local function inside a "block" structure such as `Long If...End If`, etc. The statements in *statementBlock* can contain anything except the following:

- A `Local` statement;
- An `EnterProc...ExitProc` block;
- Another local function.

Adding the `Clear` keyword causes all of the function's local variables and arrays (except the parameter variables *arg1*, *arg2* etc.) to be initialized to zeros, null strings or empty records, each time the function is called. Otherwise, the local variables and arrays will have unpredictable initial values. (You can accomplish the same effect by using the `Local` statement with the `Clear` keyword; see the `Local` statement for more information.)

The *functionName* must be unique; that is, it must be different from the name used in any other `Local Fn`, `EnterProc Fn`, `Long Fn`, `Def Fn Using` or `Def Fn <expr>` statement anywhere else in the program. If the function is to return a value, then you should specify the data type of the return value by including an appropriate type-identifier suffix at the end of *functionName*. For example, a local function which is to return a string value might be declared as follows:

```
Local Fn FullName$(idNum&)
```

The default data type of a function's return value is "long integer"; if the function is to return a long integer value, you can either include the "&" type-identifier suffix or omit it. No type-identifier suffix should be appended to *functionName* if the function does not return a value.

In order to execute the statements in *statementBlock*, your program must "call" the function using the `Fn <userFunction>` statement. The `Fn <userFunction>` statement can appear anywhere in your program, as long as the function it calls is either defined (using the `Local Fn` statement) or prototyped (using the `Def Fn <proTOTYPE>` statement) somewhere above the `Fn <userFunction>` statement. This restriction is required in order to allow your program to compile; however, the actual order of execution of your program's statements is not affected by where you place your `Local Fn`'s.

Function Parameters

Each of the parameters *arg1*, *arg2* etc. can have any of the following forms:

<i>Parameter form</i>	<i>Description</i>
<code>SimpleVar</code>	A simple numeric or string variable. Cannot be a record variable, a record field nor an array element.
<code>ptrVar As Pointer [To unType]</code>	(See below.)
<code>@longVar&</code>	<code>longVar&</code> is a simple long-integer variable.
<code>@ptrVar As Pointer [To unType]</code>	(See below.)
<code>tableau(dim1[,dim2...])</code>	<code>array</code> is a numeric or string array (not an array of records), and <code>dim1</code> , <code>dim2</code> etc. are static numeric expressions.

The parameters in the `Local Fn` statement are called the function's "formal arguments." They must not be global variables. You should not declare the formal argument variables in a `Dim` statement; they are implicitly declared by the `Local Fn` statement.

When your program calls the function, the arguments passed to it in the `Fn <userFunction>` statement are called the “actual arguments.” They must match the function’s formal arguments in number, and they must be of “compatible types” (see `Fn <userFunction>` for more information). Each time the function is called, values are assigned to its formal arguments as follows:

- If the formal argument is a `simpleVar`, the value of the actual argument is copied into `simpleVar`.
- If the formal argument is of the form `ptrVar As Pointer [To someType]`, then the actual argument should be either a record variable or a long-integer expression. In the first case, the record’s address is copied into `ptrVar`; in the second case, the long integer’s value is copied into `ptrVar`.
- If the formal argument is of the form `@longVar&`, or `@ptrVar As Pointer [To someType]`, then the actual argument must either be a variable (possibly a record variable), or a long integer expression preceded by “=”. In the first case, the variable’s address is copied into `longVar&` or `ptrVar`. In the second case, the value of the long integer expression is copied into `longVar&` or `ptrVar`.
- If the formal argument is `array(dim1 [,dim2 ...])`, then the actual argument must be the base element of an array of the same type, which has the same number of dimensions. The base element is the element in which all subscripts are set to zero. The entire array is then accessible to the local function, and (important!) any changes made to the array’s elements within the function will persist after the function exits.

If the local function has no parameters, you should omit the parentheses after *functionName*.

Passing an Array of Unknown Size

Sometimes it’s useful to write a local function which operates on an array passed in its parameter list, without knowing in advance the size of the passed array. For example, suppose you wish to write a function which sorts the elements of a long integer array, and you want it to work regardless of the declared size of the passed array.

When you declare an array as a formal parameter, FB³ ignores the value of the array’s first declared dimension in the `Local Fn` statement. For example, suppose we have a function defined like this:

```
Local Fn SetElements(anArray&(1,7), max&)
    'Set each element to 1492 in the array:
    For i& = 0 To max&
        For j = 0 To 7
            anArray&(i&,j) = 1492
        Next
    Next
End Fn
```

When we pass a long integer array to `Fn SetElements`, the passed array can have any size as its first declared dimension, as long as it has a second dimension declared as 7. For example:

```
Dim arrayOne&(1250,7), arrayTwo&(465,7)
Fn SetElements(arrayOne&(0,0), 1250)
Fn SetElements(arrayTwo&(0,0), 465)
```

Within the function, we can safely manipulate elements in the array as long as the subscripts we use don't exceed the declared dimensions of the actual array that was passed. Thus, in `Fn SetElements`, we can set the first subscript in `anArray&` to values much greater than 1, even though `anArray&` was "declared" with dimensions (1,7).

Note: you do not have equal freedom with the second, third, etc. dimensions of the parameter array. If the array is multi-dimensional, the second and subsequent dimensions must be declared with the same values in both the "formal" array parameter (in the `Local Fn` statement) and the external `Dim` statement that declares the actual passed array.

Returning a Value

If you specify an *expr* in the `End Fn` statement, the function will "return" the value of *expr*. This can be any expression which is compatible with the type-identifier suffix (if any) that appears in *functionName*. When your function "returns" a value, it means that you can reference the function (using `Fn <userFunction>`) as part of a string or numeric expression, and the function's return value will be substituted in the expression. For example:

```
maxPuppets = 6 * Fn storeCount%(x)
```

Here, if `Fn storeCount%(x)` returns a value of 7, then the value 42 will be assigned to `maxPuppets`.

The Lifespan of Local Variables

The memory space for a function's local variables is reserved when the function is called. This memory is released after the `End Fn` statement is executed. Therefore, you should never make reference to a local variable's address after the function has finished executing; in particular, you should never pass a local variable's address back to the routine that called the function. For example:

```
'DON'T DO THIS!
Local Fn myFunction&(x,y,z)
  Dim r#
  r# = Sqr(x*x + y*y + z*z)
End Fn = @r#

rAddr& = Fn myFunction&(x,y,z)
```

After the preceding is executed, `rAddr&` points to an area of memory (the old address of `r#`) which is no longer reserved, and which should not be used. On the other hand, it is permissible to pass a local variable's address into another local function. This works because the first local function has not yet finished executing when it calls the second local function. Therefore, the memory space holding the first function's local variables is still reserved intact while the second function executes.

```
'THIS IS OKAY:
Local Fn FirstFn
  Dim 255 myString$
  'Pass address of local var into another Fn:
  Fn SecondFN(@myString$)
End Fn
:
Local Fn SecondFn(strAddr&)
  BlockMove @gString$, strAddr&, Len(gString$)+1
End Fn
```

Recursive functions

You can have several functions executing simultaneously, in the sense that one function can call a second function, which can call a third, and so on. If you design your function calls in such a way that a function can call a function that is already executing, then you have a “recursive function.” The most obvious (but not the only) example of a recursive function is any function which calls itself. When that happens, we say that two (or more) “instances” of the function are executing simultaneously.

In FB³, every currently executing “instance” of a local function maintains its own private set of local variables, and they don’t interfere with the local variables of any other executing instance of that function. Calling a function recursively is very much like calling a “different” function which just happens to contain exactly the same program lines.

Although recursive functions may at first seem like a bizarre concept, they are perfectly acceptable, and often very useful. For example, here is a short program which prints all the permutations of the characters contained in a given input string; note that `Fn permute_r` calls itself. It would be very difficult to write such a program without using recursive functions.

```

'Function prototypes:
Def Fn Permute(aString$)
Def Fn permute_r(prefix$, suffix$)

Input "Enter a word: "; theWord$
Fn Permute(theWord$)
End

Local Fn Permute(aString$)
  'Prints all permutations of the letters in aString$
  Fn permute_r("", aString$)
End Fn

Local Fn permute_r(prefix$, suffix$)
  'Prints all permutations of prefix$+suffix$
  'that start with prefix$
  Long If suffix$ = ""
    Print prefix$
  Xelse
    For i = 1 To Len(suffix$)
      'Move the i-th letter of suffix$ over to newprefix$:
      newprefix$ = prefix$ + Mid$(suffix$, i, 1)
      newsuffix$ = Left$(suffix$,i-1) + Mid$(suffix$,i+1)
      'Now Print all permutations that
      'start with newprefix$
      Fn permute_r(newprefix$, newsuffix$)
    Next
  End If
End Fn

```

Returning Multiple Values

The `End Fn` statement can return only a single numeric or string expression. But many times, it's useful to have a local function which can return more than one value. The way to accomplish this is through the function's parameter list. If you give the function access to the address of some external variable or array, then the function can alter the contents at that address, effectively modifying the value of that variable or array.

There are three ways to pass an address to your function:

- If you pass an entire array (using the `array(dim1 [,dim2 ...])` syntax in the formal parameter list), then your function implicitly has access to the address of the passed array. Any changes you make to the array's elements inside your function are actually made to the external array, so the changes persist after the function exits.
- If you use the `@var` syntax in the function's formal parameter list, and specify a variable when you call the function, then the variable's address is copied into `var`. Your function can then modify the contents at that address.
- You can explicitly pass any address into a long integer or `Pointer` formal parameter.

Example:



CD Example: Local Fn.bas

See Also:

```
Fn <userFunction>; Local; @Fn; Def Fn <proTotype>
```


Locate

statement

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

Locate *h,v*

Description:

This statement moves the pen position in the current output window to text column *h* and text row *v*, based on the current font family and font size. The pen's horizontal placement is based on an "average" character width; you can't count on this position to encompass exactly *h* characters unless you are using a mono-spaced font. `Locate 0,0` places the pen at the upper-leftmost character position in the window.

See Also:

`Print@; CsrLin; Pos`

Lof**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
numRecords = Lof(deviceID [, recordLength])
```

Description:

If *deviceID* is the ID number of an open file, **Lof** returns the total number of records in the file. If *deviceID* equals `_modemPort` or `_printerPort` (and the specified device is open), **Lof** returns the total number of records currently in the device's input buffer. If there is a "partial" record at the end of the file (or input buffer), it is included in the count.

The returned record count is based on the record length given in *recordLength*, if it's specified. If this parameter is omitted, **Lof** uses the record length that was specified in the **Open** statement when the file or the port was opened. If *recordLength* is omitted and no record length was specified in the **Open** statement, a default record length of 256 is used.

To determine the total number of bytes in the file or in the serial input buffer, use `Lof(deviceID,1)`.

See Also:

`Record`; `Rec`; `Loc`; `Open`

Log

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
naturalLog# = Log(expr)
```

Description:

Returns the natural logarithm of *expr*. The natural logarithm uses the transcendental number “e” as its base. `Log` always returns a double-precision result.

`Log` is the inverse of the `Exp` function. That is: `Log (Exp (x))` equals `x`.

Note:

To find the logarithm of *expr* for an arbitrary base *n*, use this formula:

```
theLog# = Log(expr) / Log(n)
```

See Also:

`Exp`; `Log10`; `Log2`

Log10

function

✓ *Appearance***✓** *Standard***✓** *Console*

Syntax:

```
commonLog# = Log10(expr)
```

Description:

Returns the common logarithm of *expr*. the common logarithm uses “10” as its base. `Log10` always returns a double-precision result.

See Also:

`Log`; `Log2`

Log2

function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

base2Log# = **Log2**(*expr*)

Description:

Returns the base-2 logarithm of *expr*. *Log2* always returns a double-precision result.

See Also:

Log; *Log10*

Long Color

statement✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Long Color bluePart, greenPart, redPart [,ForegroundFlag]
```

Description:

Sets the foreground color or background color for the current output window. Each of the three color components can range from 0 (darkest) to 65535 (lightest), and they combine to make any conceivable shade. If you set *ForegroundFlag* to *_zTrue*, or omit the parameter, then `Long Color` sets the foreground color. If you set *ForegroundFlag* to *_false*, then `Long Color` sets the background color.

Example:

CD Example: Long Color.bas

Console behavior:

When you use the Console runtime, `Long Color` switches to the Graphics Window before executing; you can't use it to change the color in the Text Window nor on the printer. To change the color of printed output while running the Console, use the Toolbox procedure `RGBForeColor`.

Note:

`Long Color` does not immediately change the appearance of the window. If you set the foreground color, the new color will appear the next time you draw text or a `QuickDraw` shape (it won't affect the color of anything that's already drawn). If you set the background color, the new color will appear the next time you erase all or part of the window (for example, with the `Cls` statement).

You can use the Toolbox procedures `GetForeColor` and `GetBackColor` to find out the current foreground or background color for the current window.

See Also:

`Color`; `Pen`

Long Fn

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Long Fn functionName [(arg1 [,arg2...])]
      [statementBlock]
End Fn [= expr]
```

Description:

In FB^3, Long Fn is a synonym for Local Fn. Unlike FBII, variables created in a Long Fn are local.

See Also:

```
Fb <userFunction>; Local Fn; End Fn
```

Long If

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

Long If expr
    [statementBlock1]
[Xelse
    [statementBlock2]]
End If

```

Description:

The `Long If` statement marks the beginning of an “if-block,” which must be terminated with the `End If` statement. The *expr* can be either a logical expression (such as: `personCount > 17`), a numeric expression, or a string. A numeric expression is counted as “true” if it evaluates to a nonzero value. A string is counted as “true” if its length is greater than zero.

If *expr* is “true,” then only the statements in *statementBlock1* are executed, and execution then continues at the first statement after `End If`. If *expr* is “false,” then only the statements in *statementBlock2* (if any) are executed, and execution then continues at the first statement after `End If`.

statementBlock1 and *statementBlock2* may contain any number of executable statements, and may even include other “nested” if-blocks.

Note:

To conditionally execute just a single statement, consider using the `If` statement instead. To conditionally execute statement blocks based on more complex conditions, use the `Select Case` statement.

Use caution when comparing floating point values to zero or to whole numbers. The following expression may not evaluate as expected:

```
Long If x# = 1
```

In this statement, the compiler compares the value in *x#* to an integer "1". Since SANE and PPC math both use fractional approximations of numbers, the actual value of *x#*, though very close to one, may actually be something like 0.99999999 and therefore render unexpected results.

See Also:

`If; Select Case`

LPrint**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
LPrint [@(col,row)|%(h,v)] [itemList]
```

Description:

This statement sends a line of text to the printer. The `@(col,row)` and `%(h,v)` options specify where on the page the line should be printed (see the `Print` statement); if you don't specify one of these, the line is printed at the current pen position of the printing `grafPort` (this is usually just under the previously-printed line).

The `LPrint` statement is equivalent to the following group of lines:

```
Route _toPrinter
Print [@(col,row)|%(h,v)] [itemList]
Route _toScreen
```

`LPrint` is inefficient if you are printing many lines to a page, because it reroutes the output each time `LPrint` is executed. In such cases, it's better to execute a sequence of `Print` statements, with the entire sequence preceded by a single `Route _toPrinter` statement and followed by a single `Route _toScreen` statement.

Console behavior:

When you use the Console runtime, the text-position parameters `[(col,row)|%(h,v)]` are ignored.

Note:

You should execute `Clear LPrint` or `Close LPrint` in order to cause the printed page to be put out, after you have finished printing to it.

See Also:

```
Print; Clear LPrint; Close LPrint; Route
```


MachLg statement

(cpu68K only)✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
MachLg {constant|variable}[,{constant|variable}...]
```

Description:

Instructs the compiler to insert MC680x0 machine language instructions into the compiled code at the current location; these instructions are executed when the program is run.

Each *constant* is treated as a 2-byte instruction, and is inserted directly into the code stream. Typically, you will express a *constant* as a 4-digit hexadecimal number. Each *variable* is translated into an instruction which loads the variable's address into register A0.

The `MachLg` statement is valid only for a “cpu68K” compile, and you should use only MC680x0 machine language instructions in the argument list (not PPC instructions). FB³ generates an error if you attempt to use `MachLg` in a program compiled with the “cpuPPC” or “cpuFAT” options. To get around this problem, you can put the `MachLg` statement inside a “Compile Long If cpu68K” block; in that case, the `MachLg` instruction will be ignored in a PPC compile.

Note:

It's recommended that you use FutureBASIC's “Inline Assembler” syntax rather than the `MachLg` statement when you need to insert machine language instructions. Not only does this make your code easier to read, but it also allows you to insert PPC instructions as well as MC680x0 instructions.

See Also:

```
BeginAssem...EndAssem
```

MaxWindow

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
MaxWindow h, v
```

Description:

Sets a limit on how large the user can make a window by dragging its “grow box.” After `MaxWindow` is executed, the user will not be able to drag the window’s size wider than *h* pixels nor taller than *v* pixels (these dimensions refer to the window’s “content region”; they don’t include the “structure region” (frame) of the window). The `MaxWindow` statement applies to all currently open windows and all subsequently opened windows, until another `MaxWindow` statement is executed.

`MaxWindow` does not apply to windows which don’t have a grow box. Also, it does not limit the dimensions of a window which is resized by clicking its “zoom box” (see the `SetZoom` statement to learn how to do this).

If there is any currently open window (with a grow box) whose dimension(s) exceed *h* and/or *v*, `MaxWindow` will not shrink it immediately. The window will shrink the next time the user clicks on its grow box.

Note:

If you need to apply different size limits to each of several open windows, then you will need to execute `MaxWindow` each time such a window becomes active. In your dialog-event handling function, watch for the `_wndActivate` event, then execute `MaxWindow` with the appropriate parameters depending on which window has become active. See the `Dialog` function for more information.

See Also:

`MinWindow`; `Dialog` function; `SetZoom`

Maybe function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
trueOrFalse = Maybe
```

Description:

This function is a special random number generator that returns either `_zTrue (-1)` or `_false (0)`, with equal probability. Before your program calls `Maybe` for the first time, you should execute the `Randomize` statement to “seed” the random number generator.

See Also:

`Rnd;` `Randomize`

Mem**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**`info& = Mem(expr)`**Description:**

Returns information about available heap memory or about an Index\$ string array. In addition, if you specify the value `_maxAvail` in `expr`, the `Mem` function will force all purgeable resources to be removed from memory.

Specify one of the following values in `expr` to get information about the application's heap memory:

<i>expr</i>	<i>Value</i>	<i>Value returned by Mem(expr)</i>
<code>_maxAvail</code>	-1	Returns the size (in bytes) of the largest available block of contiguous free memory in the heap. Also forces all purgeable resources to be removed from memory, and may move relocatable memory blocks.
<code>_freeBytes</code>	-2	Returns the total number of free bytes in the heap. This memory may be spread over several (non-contiguous) free blocks. This number always greater than or equal to the number returned by <code>Mem(_maxAvail)</code> .

Specify one of the following values in `expr` to get information about an Index\$ string array. In this table, `indexID` represents the ID number (0 through 9) of the array you're interested in. If the specified Index\$ array has not yet been initialized (using the `Clear <index>` statement), `Mem` returns zero for all of the items in this table.

<i>expr</i>	<i>Value returned by Mem(expr)</i>
<code>IndexID + _availBytes</code>	The number of available (unused) bytes remaining in the array, following the highest element that has a string assigned to it.
<code>IndexID + _numElem</code>	The element number of the highest element in this array that has been assigned a string, plus 1. Equals zero if no element has a string assigned to it. This number is also affected by the <code>Index D</code> and <code>Index I</code> statements.
<code>IndexID + _usedBytes</code>	The number of bytes in the array that are currently occupied by strings.
<code>IndexID + _maxBytes</code>	The total number of bytes that were allocated for the array. This equals the sum of <code>Mem(indexID + _availBytes)</code> and <code>Mem(indexID + _usedBytes)</code> .
<code>Index + indxAddr</code>	The memory address of the beginning of the array.
<code>IndexID + _indxElemSz</code>	The size of each element in fixed index arrays.

See Also:

`Clear <index>;` `Index$ statement;` `Index$ function`

Menu

function

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
selectedMenu = Menu(_menuID)
selectedItem = Menu(_itemID)
```

Revision:

January 2, 2001 (Release 4)

Description:

If you have designated a menu-event handling routine in your program (using the `On Menu` statement), then `Menu(_menuID)` returns the menu number, and `Menu(_itemID)` returns the item number, of the menu item most recently selected by the user. Your menu-event handling routine should check these values each time it's called.

To give the user continual access to the menu bar, your program should execute `HandleEvents` periodically. `HandleEvents` checks for recent clicks on the menu bar, and responds by opening the menu and tracking the mouse's movement. Finally, `HandleEvents` calls your menu-event handling function if the user selects a menu item.

Menu Numbers

With the exception of the Apple Menu, the Help Menu and the Application Menu, the menus on the menu bar are numbered in increasing order from left to right. In most cases, they will be numbered consecutively starting with 1. You use the `Menu` statement to assign menu numbers to the menus your program creates.

The number of the Apple Menu equals the constant `_AppleMenu`. If your program adds new items to the Apple Menu, the `Menu` function can detect when the user selects those items. Other items in the Apple Menu are handled by the Finder, and your program can't detect when the user selects those. You use the `Apple Menu` statement to add items to the Apple Menu. (Note: If you created the Apple Menu as part of an MBar resource, then use the constant `_AppleResMenu` rather than `_AppleMenu`.)

The number of the Help Menu equals the constant `_kHMHelpMenuID`. If your program adds new items to the Help Menu, the `Menu` function can detect when the user selects those items. Other items in the Help Menu are handled by the Help Manager, and your program can't detect when the user selects those. To add new items to the Help Menu, you use the Toolbox routines `HMGetMenuHandle` and `AppendMenu` (see the `Menu` statement for an example of how to do this).

Your program can't directly detect an item selected in the Application Menu; that's handled by the Finder. However, your program can detect when another application has been brought to the front. See the `Dialog` function for more details.

Your program can also detect when the parent item that pops out a hierarchical menu is selected. This is turned on and off by a constant in the file named `UserFloatPrefs` which is located in the User Libraries folder. One (undesirable) side effect of enabling this feature is that a menu grayed by setting its title to a disabled state will produce menu events in inactive items. To use the old method (this is the default state) of ignoring hierarchical items, remark out the line in that reads...

```
_FBEnableMenuChoice = _zTrue
```

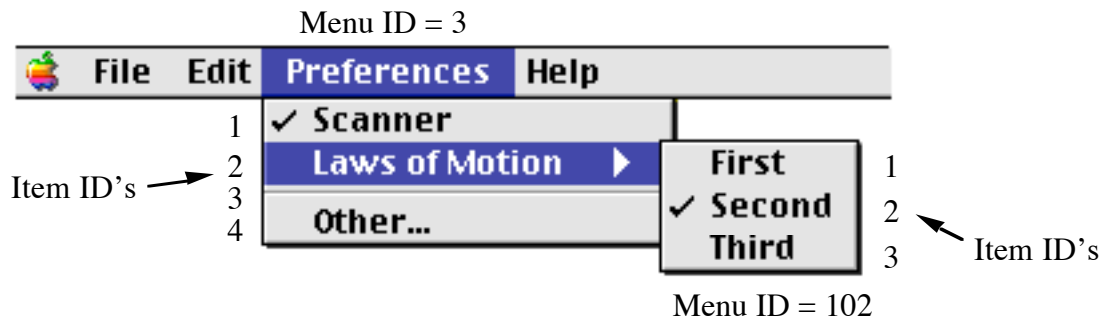
To enable the new feature, remove the remarks and allow the constant to be defined. This declaration is in the file named `UserFloatPrefs` which is located in the User Libraries folder.

Hierarchical menus have their own menu numbers that are different from their “parent” menu’s number. You can use the `Menu` function to detect selection in hierarchical menus that your program creates.

Pop-up menus are considered to be window controls (like buttons), and are therefore not detected by the `Menu` function.

Item Numbers

Menu items are numbered consecutively from top to bottom, starting with 1. Note that a grey dividing line between items has its own item number, even though it can’t be selected. It’s important to remember this when assigning and interpreting item numbers. Items within hierarchical menus are also numbered consecutively starting with 1.



See Also:

`Menu statement`; `Apple Menu`; `HandleEvents`; `On Menu Fn`

Menu

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

To create or alter a menu:

```
Menu menuID, itemID, state [, string$]
```

To unhighlight the menu bar:

```
Menu
```

Revision:

February, 2002 (Release 6)

Description:

Use this statement to do any of the following:

- Add a new menu to the menu bar.
- Enable or disable a menu.
- Add a new item to an existing menu.
- Enable or disable a menu item.
- Add or remove a checkmark from a menu item.
- Change the text of a menu item.
- Specify a hierarchical submenu to be attached to a menu item
- Unhighlight the menu bar.

To add a new menu to the menu bar:

- Set the *menuID* parameter to a number which is not already in use by an existing menu. Use a number in the range 1 through 31.
- Set the *itemID* parameter to zero.
- Set the *state* parameter either to `_enable` or `_disable`, depending on whether you want the menu to be initially enabled or dimmed (you can change this state later).
- Set the *string\$* parameter to the text that you want to appear as the new menu's title.

This creates a new empty menu (see below to learn how to add items to the menu). The value you choose for *menuID* will determine the new menu's position on the menu bar; menus are automatically positioned from left to right in increasing order of their *menuID* numbers. Almost always, you'll want to assign your menus consecutive numbers starting with 1.

To enable or disable (dim) an existing menu:

- Set the *menuID* parameter to the ID number of an existing menu.
- Set the *itemID* parameter to zero.
- Set the *state* parameter to `_enable` or `_disable`.
- Do not specify the *string\$* parameter (if you do, all the menu's items will go away!)

To add a new item to an existing menu:

- Set the *menuID* parameter to the ID number of an existing menu.
- Set the *itemID* parameter to a positive number which is not being used by any other item in the menu. This number determines the item's position in the menu; items are numbered consecutively from top to bottom starting with 1. If you "skip" an item, then either a blank space or a grey dividing line will appear in that position, depending on what version of System software you're using. Note that a grey dividing line between items has its own item ID number. You can create a grey dividing line by using the meta character "-" in the *string\$* parameter.
- Set the *state* parameter to `_enable`, `_disable` or `_checked`, depending on what you want the item's initial state to be (you can change this state later).
- Set the *string\$* parameter to the text that you want to appear in the item. Note that when you're adding a new item, certain special characters in *string\$* won't appear in the item text but have other special meanings. Consult the "Meta Characters" table below.

To enable, disable (dim), or checkmark an existing item:

- Set the *menuID* and *itemID* parameters to an existing item in an existing menu.
- Set the *state* parameter to `_enable`, `_disable` or `_checked`. Note that setting *state* to `_enable` or to `_disable` will remove any existing checkmark on the item.

To change the text of an existing item:

- Set the *menuID* and *itemID* parameters to an existing item in an existing menu.
- Set the *string\$* parameter to the desired text. Note that when you change the text of an existing item, all the characters in *string\$* will appear in the item text, and none will be interpreted as "meta characters."

To specify a hierarchical submenu to be attached to a menu item:

- Set the *menuID* parameter to the ID number of an existing menu; this is the "parent" menu which will contain the submenu.
- Set the *itemID* parameter to a positive number which is not being used by any other item in the menu. This is the "parent" item to which the submenu will be attached.
- Set the *state* parameter to the ID number of the submenu. This should be a number in the range 32 through 235 which is not being used by any other menu.
- Set the *string\$* parameter to a string which ends with these two characters: `"/" + Chr$(1B)`.
- Note: The above procedure will attach the submenu to the parent menu item, but it doesn't install the submenu. To install the submenu, you also need to call the Toolbox procedure `InsertMenu`. See the examples below.

To unhighlight the menu bar:

- Execute the `Menu` statement without any parameters. The menu bar is automatically highlighted every time the user selects a menu item, and it remains highlighted until your program unhighlights it. By unhighlighting the menu bar, your program lets the user know that the action associated with that menu item has completed.

Meta Characters

The characters in this table have special meanings when they appear in the `string$` parameter when you're adding a new menu item. Note that when you change the text of an existing item, all the characters in `string$` will appear in the item text, and none will be interpreted as meta characters. The exception to this rule is a string that starts with a minus sign. The minus sign is a flag used by most menu definitions to draw a divider line. If your item needs to contain a minus sign, you may still display the item properly if you put a space before the character.

<i>Meta character</i>	<i>Effect</i>
<code>;</code>	When it appears by itself, “ <code>;</code> ” creates a grey dividing line. When it appears as a delimiter in a list (e.g., “ <code>item1;item2</code> ”), each of the items in the list becomes a separate menu item. You can use this fact to add several new menu items with just a single <code>Menu</code> statement.
<code>(</code>	When it appears in an item that follows a semicolon, “ <code>(</code> ” initially disables (dims) the item.
<code>/</code>	The character following “ <code>/</code> ” becomes a command-key equivalent for the menu item. Or, if the character following “ <code>/</code> ” is <code>Chr\$(&1B)</code> , it indicates that this menu item has a submenu.
<code>!</code>	When “ <code>!</code> ” appears in an item that follows a semicolon, the character following “ <code>!</code> ” is displayed as a “mark” on the left side of the menu item.
<code>^</code>	The number (1 through 9) following “ <code>^</code> ” is added to 256 to get an icon resource number. The corresponding icon is displayed on the left side of the menu item.
<code>-</code>	Creates a grey dividing line. Any other characters in the item string are ignored.
<code><</code>	The letter following “ <code><</code> ” is interpreted as a text attribute to be applied to the menu item. Use one of the following letters: <div style="display: flex; justify-content: space-around; margin-top: 5px;"> <code>B</code> = Bold <code>O</code> = Outlined <code>U</code> = Underlined </div> <div style="display: flex; justify-content: space-around; margin-top: 5px;"> <code>I</code> = Italic <code>S</code> = Shadowed </div>

Creating Hierarchical Menus

You can use the following function to add a new menu item and attach a new hierarchical menu to it. You should set `childMenuID` to some number in the range 32 through 235 which is not being used by any existing menu.

```

Local Fn MakeHierMenu (parentMenuID, parentMenuItem, ~
                        itemString$, childMenuID)

    title$ = "!" + Chr$(childMenuID) + itemString$ + "/" + Chr$( &1B)
    Menu parentMenuID, parentMenuItem, , title$
    Call InsertMenu (Fn NewMenu (childMenuID, ""), -1)
End Fn

```

After you have called `Fn MakeHierMenu`, you can use the `Menu` statement to add new items to the hierarchical menu (set the `menuID` parameter to the value of `childMenuID`).

Items in the Apple Menu

You should use the `Apple Menu` statement to add items to the top of the Apple Menu. After adding these items, you can use the `Menu` statement (with the `menuID` parameter set to `_appleMenu`) to alter the items (for example to enable or dim them).

Items in the Help Menu

You can add items to the bottom of the Help Menu by getting a handle for the Help Menu and then calling the `AppendMenu` procedure. You also need to find out the item number of your first Help item for use by your menu event handler (any existing items are handled by the Help Manager):

```
Dim As Int OSErr, @ firstCusTomHelpItem
Dim As Handle @ hmHandle

#If carbonlib
    OSErr = Fn HMGetHelpMenu(hmHandle, firstCusTomHelpItem)
#Else
    OSErr = Fn HMGetHelpMenuHandle(hmHandle)
    firstCusTomHelpItem = Fn CountMenuItems(hmHandle)+1
#EndIf

Call AppendMenu(hmHandle, "My Help")
```

After adding items to the Help Menu, you can use the `Menu` statement (with the `menuID` parameter set to `_kHMHelpMenuID`) to alter the items.

Note: Do not use the `Menu` statement to add new items to the Help Menu; use `AppendMenu` instead.

Removing Menus

Call the `DeleteMenu` procedure to remove a menu created by the `Menu` statement:

```
Call DeleteMenu(menuID)
```

This may cause other menus in the menu bar to slide to the left to fill the gap; however, they still retain their original menu ID numbers.

Removing Menu Items

To remove all the items from a menu you created, use the `Menu` statement, specifying zero in the `itemID` parameter, and specifying a menu title in the `string$` parameter.

To remove an individual item, use the `GetMenuHandle` function and the `DeleteMenuItem` procedure:

```
Call DeleteMenuItem(Fn GetMenuHandle(menuID), itemID)
```

Note that this will renumber any items below the deleted item, as they move up to fill in the gap. Menu item numbers are always numbered consecutively starting with 1.

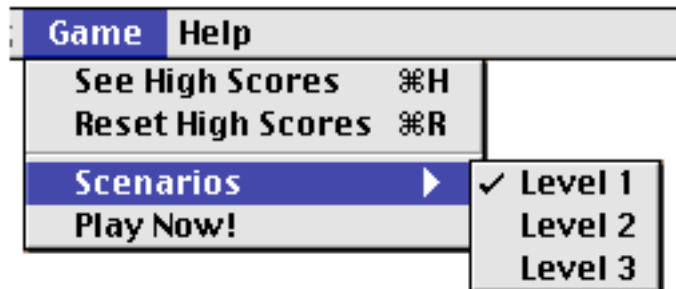
Example:

The following lines create a complete menu which also contains a hierarchical menu. This example makes use of the MakeHierMenu function defined above.

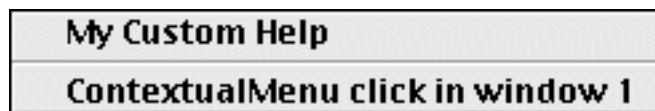
```

Menu 3,0,_enable,"Game"
Menu 3,1,_enable,"See High Scores/H"
Menu 3,2,_enable,"Reset High Scores/R"
Menu 3,3,_disable,"-"
Fn MakeHierMenu(3,4,"Scenarios",100)
'Items in hierarchical menu:
Menu 100,1,_checked,"Level 1"
Menu 100,2,_enable,"Level 2"
Menu 100,3,_enable,"Level 3"
'It takes two Menu statements to include a
'special character like "!" in the text:
Menu 3,5,_enable,"dummy"   'This adds the item
Menu 3,5,_enable,"Play Now!" 'This alters the item

```

**Contextual Menus**

At about the time that the Appearance Manager came on to the scene, programmers began to use contextual menus. A contextual menu appears when the user clicks at a specific area in a window while holding down the control key. When this type of action takes place, you will receive (Appearance Manager Runtime only) a `Dialog(0)` message of `_cntxtMenuClick`. `Dialog(_cntxtMenuClick)` will be the window number of the window. At this point you may need to react by showing a menu under the cursor.



The following function builds and displays a menu and might be called in reaction to a contextual menu click.

```

Local Fn DoContextMenu( wNum As Long )
  Dim @ selectionType As Long
  Dim @ menuID As Short
  Dim @ menuItem As Short
  Dim mHndl As Handle
  Dim err As OSStatus
  Dim helpItemString As Str255
  mHndl = Fn NewMenu(255, "X")
  Long If mHndl
    InsertMenu( mHndl, -1 )
    AppendMenu( mHndl, -
      "ContextualMenu click in Window" + Str$( wNum ) )
    helpItemString = "My Custom Help"
    err = Fn ContextualMenuSelect( mHndl, -
      #gFBTheEvent.where, _nil, _kCMHelpItemNoHelp, -
      @helpItemString, #_nil, @selectionType, -
      @menuID, @menuItem )
    /*
      In this function, we don't actually do anything with
      the selectionType, menuID, or menuItem returned, but
      we could react to it right here
    */
    DisposeMenu( mHndl )
  End If
End Fn

```

See Also:

Menu <resource>; Menu **function**; On Menu **Fn**; Apple Menu; Def CheckOneItem

Menu <resource>**statement**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
Menu resID%, {_resMenu|_resSubMenu}, state [, resType$]
```

Description:

Use this form of the `Menu` statement to install a menu which is described in a “MENU” resource. The menu’s items can either be defined in the “MENU” resource, or they can consist of a list of available named resources of a particular type (such as font resources).

The `resID%` parameter should specify the resource ID number of a “MENU” resource in a currently open resource file (usually in your application’s resource fork). In most cases, this will also equal the menu’s ID number (although the menu’s ID number is actually defined within the “MENU” resource).

If you specify `_resMenu`, the menu will be installed on the menu bar. In this case, you should use a menu with a menu ID number in the range 1 through 31, which is different from the ID number of any existing menu.

If you specify `_resSubMenu`, the menu won’t be installed on the menu bar, but will be added to an internal “menu list.” Use this option when you want to install the menu as a hierarchical menu or a pop-up menu. To install a hierarchical resource menu, you must also use the Toolbox procedure `InsertMenu`. The following illustrates how this may be done:

```
resID% = 130
menuID% = 130

'Put resource MENU On internal "Menu list":
Menu resID%, _resSubMenu, _enable

'Set this menu's "parent item":
Menu parentMenuID,parentItemID,menuID%,title$+"/"+Chr$(&1B)

'Attach the resource sub-menu to the parent menu:
Call InsertMenu(Fn GetMenuHandle(menuID%), -1)
```

If you use the `_resSubMenu` option, you should use a menu with a menu ID number in the range 32 through 235, which is different from the ID number of any existing menu.

The `state` parameter specifies whether the menu should initially be enabled or disabled (dimmed). Set this parameter either to `_enable` (1) or to `_disable` (0).

If you omit the *resType\$* parameter, the resource menu's title and menu items will be displayed as defined in the "MENU" resource. If you specify the *resType\$* parameter, it should be a 4-character string indicating a resource type. In this case, the menu's items will consist of a list of all available named resources of the indicated type. For example, if you specify a *resType\$* value of "FOND", the menu will list the names of all available fonts. The menu's title is still taken from the "MENU" resource.

Note:

You can use a program like ResEdit to create "MENU" resources.

See Also:

Menu statement; Menu function

Menu Preferences

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
Menu Preferences menuID, itemID
```

Revision:

February, 2002 (Release 6)

Description:

In OS-X, preferences are no longer accessible from the bottom item of the Edit menu. They are traditionally moved to the application menu. This means that you would have to do a large amount of coding to insure that your program followed the proper guidelines for both System 9 and OS-X. Instead, you may use the `Menu Preferences` statement. If you are operating in System 9, this command does nothing. If your application is being run in OS-X, the preference menu is moved to the application menu. Further, if the user selects the preference item, the menu choice is converted to the specified *menuID* and *itemID* so that your program can react without special coding.

A full example of using the new `Preferences` commands can be found on page 453.

Note:

This will not work properly if the *itemID* is anything other than the last item of the edit menu. FB must delete the old preference item and this would change menu IDs in your program if other selections followed it.

See Also:

`Put Preferences; Get Preferences; Kill Preferences`

Mid\$ and Mid\$\$

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
subString$      = Mid$(string$, startPos [, numChars])
subContainer$$ = Mid$$(container$$, startPos [, numChars])
```

Revision:

June 15, 2000 (Release 3)

Description:

This function returns a substring or subcontainer of *string\$* or *container\$\$*, consisting of characters which begin at position *startPos* within *string\$* or *container\$\$*. If you specify *numChars*, then a maximum of *numChars* characters are returned; otherwise, all the characters from *startPos* to the end of *string\$* or *container\$\$* are returned. If *startPos* is less than 1, then it's treated as 1. If *startPos* is greater than the length of *string\$* or *container\$\$*, then a null (zero-length) string is returned.

Note:

You may not use complex expressions that include containers on the right side of the equal sign. Instead of using:

```
c$$ = c$$ + Mid$$(a$$,10)
```

Use:

```
c$$ += Mid$$(a$$,10)
```

Example:

```
Print Mid$("Rick Brown", 2, 3)
myContainer$$ = "Rick Brown"
Print Mid$(myContainer$$, 2, 3)
Print Mid$("Rick Brown", 6)
```

Program output:

```
ick
ick
Brown
```

See Also:

Mid\$ statement; Left\$; Right\$; InStr

Mid\$ and Mid\$\$

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

Mid$(stringVar$, startPos, numChars) = replaceString$
Mid$$(container$$, startPos, numChars) = replaceString$|contnr$$

```

Revision:

May 30, 2000 (Release 3)

Description:

This statement updates *stringVar\$* (which must be a string variable) or *container\$\$* (a container variable), deleting a subpart from *stringVar\$* or *container\$\$* and replacing it with an equal number of characters from the left side of *replaceString\$*. The subpart to be replaced begins at position *startPos* within *stringVar\$* or *container\$\$*. In the following code fragments, containers and strings work the same. The number of characters replaced equals the smallest of these quantities:

- *numChars*
- `Len(replaceString$)`
- `Len(stringVar$) - startPos + 1`

Under the following circumstances, **Mid\$** does nothing:

- When *stringVar\$* or *replaceString\$* is empty;
- When *startPos* is less than 1 or greater than `Len(stringVar$)`;
- When *numChars* is less than 1.

Note:

You may not use complex expressions that include containers on the right side of the equal sign.

Example:

```

x$ = "abcdefgh"
y$ = "abcdefgh"
z$ = "abcdefgh"
Mid$(x$,2,3) = "1234" : Print x$
Mid$(y$,2,5) = "1234" : Print y$
Mid$(z$,7,4) = "1234" : Print z$

```

Program output:

```

a123efgh
a1234efgh
abcdef12

```

See Also:

`Mid$` function; `Left$`; `Right$`; `InStr`

MinWindow

statement✓ *Appearance*✓ *Standard*✗ *Console***Syntax:****MinWindow** *h, v***Description:**

Sets a limit on how small the user can make a window by dragging its “grow box.” After `MinWindow` is executed, the user will not be able to drag the window’s size narrower than *h* pixels horizontally nor shorter than *v* pixels vertically (these dimensions refer to the window’s “content region”; they don’t include the “structure region” (frame) of the window). The `MinWindow` statement applies to all currently open windows and all subsequently opened windows, until another `MinWindow` statement is executed.

`MinWindow` does not apply to windows which don’t have a grow box. Also, it does not limit the dimensions of a window which is resized by clicking its “zoom box” (see the `SetZoom` statement to learn how to do this).

If there is any currently open window (with a grow box) whose dimension(s) are smaller than *h* and/or *v*, `MinWindow` will not expand it immediately. The window will expand the next time the user clicks on its grow box.

Note:

If you need to apply different size limits to each of several open windows, then you will need to execute `MinWindow` each time such a window becomes active. In your dialog-event handling function, watch for the `_wndActivate` event, then execute `MinWindow` with the appropriate parameters depending on which window has become active. See the `Dialog` function for more information.

See Also:`MaxWindow`; `Dialog` function; `SetZoom`

Mki\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
string$ = Mki$(intExpr)
```

Description:

Mki\$ (“MaKe Integer string”) returns a string which has the same internal bit pattern as *intExpr*; each character in the returned string will represent 8 bits from *intExpr*. The returned string will have a length of 1, 2 or 4 characters, depending on which of DefStr Byte, DefStr Word or DefStr Long is currently in effect. If DefStr Byte is in effect, you should make sure that *intExpr* is within the range of numbers that can be expressed in a single byte; similarly, if DefStr Word is in effect, you should make sure that *intExpr* is within the range of numbers that can be expressed in a “word”-length (2-byte) integer.

Mki\$ is useful for translating the 4-letter file types, creator codes, resource types, etc. that are frequently used in MacOS Toolbox routines. These codes are typically transmitted in the form of long-integer values; by using the Mki\$ function you can translate these long integers into strings for display purposes (be sure to set DefStr Long before doing this).

If DefStr Byte is in effect, Mki\$ returns the same thing as the Chr\$ function.

Note:

When DefStr Long is in effect and 4-character strings and long-integers are being converted, Mki\$ is essentially the inverse of the Cvi function. Note, however, that the behavior of Cvi does not depend on the the current setting of DefStr Byte/Word/Long.

See Also:

Cvi; DefStr Byte/Word/Long; Chr\$; Str\$; Val

Mod

operator

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

remainder = *expr* **Mod** *modulus*

Description:

The `Mod` operator subtracts from `Abs(expr)` the largest multiple of `Abs(modulus)` which is less than or equal to `Abs(expr)`, and returns the result as remainder. If *expr* is negative, then a negative result is returned in remainder.

Note that if *expr* and *modulus* are both integers, the result of `Mod` is just the remainder of the integer division operation *expr* / *modulus*.

See Also:

Appendix D: *Numeric Expressions*

Mouse(_down)**function**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
butTonStatus = Mouse(_down)
```

Description:

If your program does not do any kind of event-trapping using the `HandleEvents` statement, then you can use the `Mouse(_down)` function to determine whether the mouse button is currently down. In these circumstances, `Mouse(_down)` returns `_zTrue` if the button is down, or `_false` otherwise.

The `Mouse(_down)` function will not work if your program traps events using `HandleEvents`. Since most well-designed programs trap events this way, the `Mouse(_down)` function is probably of limited usefulness. See the `Mouse <event>` functions to learn how to respond to mouseclicks using event trapping.

Note:

You can also use the Toolbox function `Fn Button` to determine whether the mouse is currently down. `Fn Button` works regardless of whether your program uses event trapping.

See Also:

```
Mouse <position>; Mouse <event>; On Mouse
```

Mouse <event>**functions**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
clickType = Mouse(0)
locationInfo = Mouse(locationType)
```

Revision:

February, 2002 (Release 6)

Description:

If you have designated a mouse-event handling routine using the `On Mouse` statement, then the `Mouse <event>` functions return information about a mouseclick event. Your mouse-event handling routine should check the `Mouse(0)` function, and possibly the `Mouse(locationType)` functions, each time your routine is called.

The `Mouse <event>` functions will not report a mouseclick that occurs inside an active control (such as a button or scrollbar), or in an edit field or picture field, or anywhere outside the active window's content region. Such mouseclicks are handled by other routines, such as your dialog-event handling routine (see the `Dialog` function), or your menu-event handling routine (see the `Menu` function).

Mouse(0) function

The `Mouse(0)` function indicates whether a single, double or triple-click occurred. It will usually return one of the following values:

<i>Mouse(0)</i>	<i>Value</i>	<i>Description</i>
<code>_click1nDrag</code>	-1	single click, and mouse is still down.
<code>_click2nDrag</code>	-2	double click, and mouse is still down.
<code>_click3nDrag</code>	-3	triple click, and mouse is still down.

In rare cases, the user may have time to both click the mouse and release it before your program detects the click. This can happen, for example, if your program runs a long time between successive calls to `HandleEvents`. In that case, `Mouse(0)` may return one of the following values:

<i>MOUSE(0)</i>	<i>Value</i>	<i>Description</i>
<code>_click1</code>	1	single click, and mouse is already released.
<code>_click2</code>	2	double click, and mouse is already released.
<code>_click3</code>	3	triple click, and mouse is already released.

If you just want to detect the click, and you don't care whether the user released the mouse button before your mouse-event handling routine was called, then your routine can just check `Abs(Mouse(0))`, which will always return 1, 2 or 3.

Mouse(locationType) functions

To detect where the mouse pointer was at the instant it was clicked, call the `Mouse(_lastMHorz)` and `Mouse(_lastMVert)` functions within your mouse-event handling routine. The values returned by `Mouse(_lastMHorz)` and `Mouse(_lastMVert)` are usually the same as those returned by `Mouse(_horz)` and `Mouse(_vert)` (see the `Mouse <position>` functions), but they may be different, especially if the mouse is being moved quickly.

If `Mouse(0)` returns a positive value (indicating that the mouse was both clicked and released before your mouse-event handling routine was called), then you may also be interested in the values returned by `Mouse(_releaseHorz)` and `Mouse(_releaseVert)`. These values tell you where the mouse pointer was at the instant the mouse button was released. If `Mouse(0)` returned a negative value, then `Mouse(_releaseHorz)` and `Mouse(_releaseVert)` are meaningless.

Mouse Window (Appearance Manager)

A new selector helps your program determine where the mouse is located:

```
wndNum = Mouse(_mouseWindow)
```

...will return the FB window reference number of the window over which the mouse is positioned. The window does not need to be active when this is used.

Click Sequencing

FB^3 reports a mouseclick event as soon as it can after the mouse button has been pressed down. If the user executes a double-click, FB^3 interprets it first as a single-click event and then (once the second click happens) as a double-click event. Both "events" will be reported to your mouse-event handling routine. Similarly, if the user executes a triple-click, FB^3 will first report a single-click event, then a double-click event, and finally a triple-click event.

You should take this into account when writing your mouse-event handling routine. The example program "DoubleClick.bas" handles single-clicks and double-clicks; like most well-designed programs, its interface is designed so that the effects of a single-click are included in the effects of a double-click.

Waiting for the Mouse Up

In most cases, FB^3 will call your mouse-event handling routine while the mouse button is still being held down. But in some situations, your routine may need to track the mouse's motion until the button is released. You can use the Toolbox function `Fn StillDown` to determine when the user releases the mouse button. See the example program, "StillDown.bas".

Example:



CD Example: `DoubleClick.bas`; `StillDown.bas`

See Also:

```
Mouse(_down); Mouse <position>; On Mouse; HandleEvents
```

Mouse <position>**functions**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```

horzPosition = Mouse(_horz)
vertPosition = Mouse(_vert)

```

Description:

If your program does not do any kind of event-trapping using the `HandleEvents` statement, then `Mouse(_horz)` and `Mouse(_vert)` return the mouse pointer's current horizontal and vertical pixel coordinates, relative to the upper-left corner of the current output window.

If your program does use `HandleEvents`, then the `Mouse <position>` functions indicate where the mouse was the last time `Mouse(0)` was accessed (see the `Mouse <event>` functions).

Example:

```

Window 1
Print "Move the mouse around. Press Cmd-period to quit."
Do
  HandleEvents
  dummy = Mouse(0) 'to activate the Mouse <position> Fn's
  xNew = Mouse(_horz): yNew = Mouse(_vert)
  Long If xNew <> xOld Or yNew <> yOld
    Locate 0,1: Cls Line
    Print xNew, yNew
    xOld = xNew: yOld = yNew
  End If
Until _false

```

Note:

Use the `Mouse <event>` functions to determine the mouse's position at the instant it was clicked.

See Also:

```
Mouse(_down); Mouse <event>
```


Nand

operator

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

result& = *exprA* {**Nand** | ^&} *exprB*

Description:

Expression *exprA* and expression *exprB* are each interpreted as 32-bit integer quantities. The Nand operator sets each bit in *result* when the bit in *exprA* is set and the corresponding position in *exprB* is cleared. This can be thought of as a Not And expression. The result is another 32-bit quantity; each bit in the result is determined as follows:

<i>Bit value in exprA</i>	<i>Bit value in exprB</i>	<i>Bit value in result&</i>
0	0	0
1	0	1
0	1	0
1	1	0

See Also:

And, Nor, Not; Xor; Or; Appendix D: *Numeric Expressions*

Name	statement (Obsolete)
------	----------------------

This used to be a synonym for the `Rename` statement. It is now obsolete.

Next

statement

See the `For` statement.

Nor

operator

✓ Appearance

✓ Standard

✓ Console

Syntax:

```
result& = exprA {Nor | ^|} exprB
```

Description:

Expression *exprA* and expression *exprB* are each interpreted as 32-bit integer quantities. The *Nor* operator sets each bit in *result* when the corresponding bits in both *exprA* and *exprB* are cleared or when the bit in *exprA* is set and the corresponding bit in *exprB* is cleared. This can be thought of as a *Not Or* expression. The result is another 32-bit quantity; each bit in the result is determined as follows:

Bit value in <i>exprA</i>	Bit value in <i>exprB</i>	Bit value in <i>result&</i>
0	0	1
1	0	1
0	1	0
1	1	1

See Also:

And, Nand, Not; Xor; Or; *Appendix D: Numeric Expressions*

Not operator

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
value = Not expr
```

Description:

The `Not` operator interprets *expr* as an integer, and returns another integer in whose internal bit pattern all the bits are flipped to their opposite state (i.e., all 1's are changed to 0; and all 0's are changed to 1). Coincidentally, because of the way that integers are stored in FB^3, the value returned by `Not expr` equals: $-(expr + 1)$.

One common use for `Not` is to reverse the sense of an expression whose value equals `_zTrue` (−1) or `_false` (0). Note that `(Not _zTrue)` returns `_false`, and `(Not _false)` returns `_zTrue`. You must be careful when using `Not` with “true” values other than −1. For example:

```
testValue = 35
If testValue Then Beep      'This produces a Beep
If Not testValue Then Beep 'But so does this!
```

This program produces two beeps, because in the second `If` statement, “`Not testValue`” produces the value −36, which is still interpreted as “true” by the `If` statement.

Another common use for `Not` is to help you set or reset individual bits in a bit pattern. For example:

```
pattern& = pattern& And Not Bit(7)
```

This sets bit 7 in `pattern&` to zero, and leaves all of `pattern&`'s other bits alone.

See Also:

`And`; `Or`; `Xor`

Oct\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
octalString$ = Oct$(expr)
```

Description:

This function is a string of octal (base-8) digits which represent the integer value of *expr*. The returned string will consist of either 3, 6 or 11 characters, depending on which of `DefStr Byte`, `DefStr Word` or `DefStr Long` is currently in effect. Note that if the value of *expr* is too large to fit in the currently selected `DefStr` size, the string returned by `Oct$` will not represent the true value of *expr*.

In FB³, integers are stored in standard “2’s-complement” format, and the values returned by `Oct$` reflect this storage scheme. You need to keep this in mind when interpreting the results of `Oct$`, especially when *expr* is a negative number. For example: `Oct$(-3)` returns “775” when `DefStr Byte` is in effect; “777775” when `DefStr Word` is in effect; and “7777777775” when `DefStr Long` is in effect.

Note:

To convert a string of octal digits into an integer, use the following technique:

```
intVar = Val("&o" + octalString$)
```

`intVar` can be a (signed or unsigned) byte variable, short-integer variable or long-integer variable. See Appendix C: *Data Types and Data Representation*, to determine the range of values that can be stored in different types of integer variables.

See Also:

```
Hex$; Bin$; DefStr Byte/Word/Long; Val&
```

OffsetOf

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
byteOffset = OffsetOf(fieldName In {recordType|trueRecVar})
byteOffset = OffsetOf(const [In pseudoRecVar])
```

Description:

Use this function to find where a particular field begins within a record. `OffsetOf` returns the field's offset as a number of bytes past the beginning of the record.

In the first syntax, *recordType* is the name of a “true record” type as defined in a `Begin Record` statement; *trueRecVar* is a variable declared as a “true record” type; *fieldName* is the name of a field within that “true record” type.

In the second syntax, *const* is a symbolic integer constant, without its leading underscore character. Typically, this will be a constant which was defined within a `Dim Record...Dim End Record` block. *pseudoRecVar.recLen* is a “pseudo-record” variable declared as `Dim pseudoRecVar.recLen`. When you use the second syntax, `OffsetOf` just returns the value of *const*. Because a symbolic constant always has a global scope, the `In pseudoRecVar` clause is not required.

The value passed as *fieldName* is seen by the compiler as a constant. You do not use type designator suffixes like `$`, `&`, `#`, etc.

See Also:

`SizeOf`; `TypeOf`; `Begin Record`; `Dim Record`; [Appendix C: Data Types and Data Representation](#)

On AppleEvent

statement

✓ *Appearance*

✓ *Standard*

✗ *Console*

Syntax:

```
On AppleEvent (eventType&,eventClass&)-
    {Fn userFunction|Gosub{lineNumber|"stmtLabel"}}
```

Revision:

May, 2001 (Release 5)

Description:

AppleEvents are used for communication between applications. (In dealing with Apple Events, you will often see applications referred to as processes.) Use this FB statement to set up and track specific types of Apple Events. Event types and classes that are to be tracked are part of the `On AppleEvent` parameters.

Types and Classes

Apple events are divided into general groups called Types. Those groups are subdivided into specific classes. As an example, you may have heard of Apple core events. There are events which every application should understand and handle. The class for this group is called `_coreEventType`. the long integer constant for this type is `_"core"`. Either of these two forms of address is acceptable in FB, but only one (`_"core"`) is case sensitive.

If you wanted to deal with core events, you would have to tell the Apple Event Manager which of four classes might be of interest. As an example, you might want to know if the Finder (or some other process) sent a message to you to open a document. In such a case the class would be `_kAEOpenDocuments` or `_"odoc"`.

An event handler set up for this type and class would be entered as follows:

```
On AppleEvent(_coreEventType,_kAEOpenDocuments) Fn myODocHandler
```

FB recognizes a special event class of `_"TEXT"`. When this class is encountered, the contents of the event parameter may be more easily accessed using the `AppleEventMessage$` function.

The following simple example shows how your `On AppleEvent` handler might look for incoming text.

```
Local Fn doAppleEvent
    Print "Message received:"
    Print AppleEventMessage$
End Fn

On AppleEvent(_"buba",_"TEXT") Fn doAppleEvent
```

It is often convenient to pass large blocks of data via Apple Events as handles. FB processes this automatically by pulling the data from the Apple Event and transferring it to your application as a handle. FB always *copies* the data, so when you are finished, you must manage disposal on your own. The following fragment shows how a handle is received and disposed. To show that the data was accepted, this routine prints the first fifty characters of the handle as ASCII text.

```

Local
Dim AERec As AERecord
Dim @OSErr
Dim dataHandle As Handle
Dim n
Local Fn doAppleEvent
    OSErr = Fn AEGetParamDesc(gFBAEEvent&,-
        _keyDirectObject, _"CLAS", AERec)
    dataHandle = AERec.dataHandle
    Cls
    Print "Rec'd at ";Time$
    For n = 1 To 50
        Print Chr$(Peek([dataHandle]+n));
    Next
    Call DisposeHandle(dataHandle)
End Fn

```

When this handler was set up, the *eventClass* parameter for `On AppleEvent` was `_"CLAS"`. You can see that it was required for use in `Fn AEGetParamDesc`.

Note:

Some handlers (like the example above) are already built in to most of the FB runtimes. In order to override a behavior, you will first have to use `Kill AppleEvent`.

Only one Apple Event vector is allowed. If you create multiple `On AppleEvent` vectors, each one overwrites the previous, so the last vector will be used. To handle multiple eventClasses and/or types in the one application, set up a vector for each, pointing all to the one handler. The handler can determine which class it has received by examining the global variable `gFBAEType&`. (See `Kill AppleEvent` for the proper method of replacing an event vector.)

Example:



CD Example: AppleEvents folder

See Also:

`SendAppleEvent`, `HandleEvents`, `AppleEventMessage$`, `GetProcessInfo`,
`Kill AppleEvent`

On Break

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
On Break {Fn userFunction|Gosub{lineNumber|"stmtLabel"}}
```

Description:

This statement designates a particular function or subroutine as a break-event handling routine. A break-event handling routine is called in response to a ⌘-period keypress by the user.

After a ⌘-period keypress event occurs, FB^3 does not call your designated routine immediately. Instead, your program continues executing until it reaches a `HandleEvents` statement; at that time, `HandleEvents` will call your designated routine.

If the user presses ⌘-period but you have not designated any break-event handling routine, your program continues executing until a `HandleEvents` is reached; FB^3 then displays an alert asking the user whether to continue or to stop. If the user elects to stop, FB^3 next calls your “stop-event” handling routine, if any (see `On Stop`); and then your program stops. If the user elects to continue, the program continues executing at the first statement following the `HandleEvents`.

One common use for a break-event handling routine is to prevent the user from stopping your program by means of ⌘-period. Your routine can be as simple as this:

```
Local Fn DoBreak
End Fn
```

This function does nothing, but if you designate it in an “`On Break Fn DoBreak`” statement, then ⌘-period keypresses will be “routed” to this function rather than causing the “Stop or Continue” alert to be displayed.

Note:

If you use the `On Break Fn userFunction` syntax, then `userFunction` must refer to a function which was defined or prototyped at an earlier location in the source code. Your break-event handling function should not take any parameters, nor return a result.

See Also:

`HandleEvents`; `On Stop`; `Tron X`

On Dialog

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
On Dialog {Fn userFunction|Gosub{lineNumber|"stmtLabel"}}
```

Description:

This statement designates a particular function or subroutine as a dialog-event handling routine. A dialog-event handling routine is called in response to a number of different kinds of user actions and internal events; see the `Dialog` function for more information.

After a dialog event occurs, FB³ does not call your designated routine immediately. Instead, your program continues executing until a `HandleEvents` statement is reached. At that time, `HandleEvents` will call your designated routine once for each dialog event that occurred; your designated routine should examine the `Dialog(0)` and `Dialog(evnt)` functions to get information about the event. If you have not designated any dialog-event handling routine, FB³ ignores events of this kind.

Note:

If you use the `On Dialog Fn userFunction` syntax, then *userFunction* must refer to a function which was defined or prototyped at an earlier location in the source code. Your dialog-event handling function should not take any parameters, nor return a result.

See Also:

`Dialog` function; `Dialog` statement; `HandleEvents`

On Edit

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
On Edit {Fn userFunction|Gosub{lineNumber|"stmtLabel"}}
```

Description:

This statement designates a particular function or subroutine as an edit-event handling routine. An edit-event handling routine is called in response to a keypress, if there is an active editable edit field in the current output window.

After such a keypress event occurs, FB^3 does not call your designated routine immediately. Instead, your program continues executing until the next `HandleEvents` statement is reached; at that time, FB^3 calls your designated routine once for each keypress event that occurred. FB^3 calls your routine without displaying the new character in the edit field.

Your edit-event handling routine should examine the `TEKey$` function to determine which key was pressed. To apply that character (or some other character of your choice) to the edit field, your routine should execute the `TEKey$` statement.

If you have not designated any edit-event handling routine, then `HandleEvents` applies the keypress directly to the current edit field.

The most common use for an edit-event handling routine is to inhibit or alter the display of certain kinds of characters in the edit field. In most cases, you will want the following keys to be processed normally:

- `Chr$(8)` (delete)
- `Chr$(28)` (left arrow)
- `Chr$(29)` (right arrow)
- `Chr$(30)` (up arrow)
- `Chr$(31)` (down arrow)
- `Chr$(13)` (Return (if edit field type allows Return's))

Example:

This program limits the entered contents of the edit field to a number of not more than four digits.

```

Def Fn DoEdit ' (proTOTYPE)
Window 1
Text _sysFont, 12
Edit Field 1, "", (10,10)-(100,30), _framedNoCR
On Edit Fn DoEdit
Do
  HandleEvents
Until _false
End
'-----
Local Fn DoEdit
  theKey$ = TEKey$
  efID = Window(_efNum)
  Select Case theKey$
    Case "0","1","2","3","4","5","6","7","8","9"
      Long If Len(Edit$(efID)) < 4
        TEKey$ = theKey$ 'display the digit
      Xelse
        Beep 'Too many digits!
      End If
    Case Chr$(8),Chr$(28),Chr$(29),Chr$(30),Chr$(31)
      TEKey$ = theKey$ 'Apply the cursor movement
    Case Else
      Beep 'Illegal character!
  End Select
End Fn

```

Note:

If you use the `On Edit Fn userFunction` syntax, then *userFunction* must refer to a function which was defined or prototyped at an earlier location in the source code. Your edit-event handling function should not take any parameters, nor return a result.

The following kinds of keypresses do not trigger a call to your edit-event handling routine:

- Menu Command-key equivalents;
- ⌘-period;
- Arrow keys in a direction that the insertion point can't move (these generate Dialog events instead);
- Modifier keys (e.g. Shift, Control, etc.) by themselves.

If there is no active editable edit field in the current window or if you are running under the Appearance Compliant runtime, keypresses are treated as Dialog events (see the `_evKey` event in the Dialog function).

See Also:

TEKey\$ function; TEKey\$ statement; Dialog function; HandleEvents

On Error Fn/Gosub

statements

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
On Error {Fn userFunction|Gosub{lineNumber|"stmtLabel"}}
```

Revision:

June, 2001 (Release 5)

Description:

This statement designates and enables the routine that FB will call when certain kinds of errors occur. There may only be one `On Error` vector. If you use a second call to `On Error Fn`, the new routine replaces the old version in subsequent calls. However you can deactivate or reactivate error trapping as often as you need. Using the `On Error End`/`On Error Return` statements you can switch between the default behavior and your error handler.

Example:

```
Local Fn HandleFileError
    Print "Sorry. Something has gone wrong."
    Stop
End Fn

On Error Fn HandleFileError
Open "I",#1,"this file does not exist"
```

Note:

If you use the `On Error Fn userFunction` syntax, then *userFunction* must refer to a function which was defined or prototyped at an earlier location in the source code. Your error handling function should not take any parameters, nor return a result.

See Also:

`On Error End`; `On Error Return`; `Error`; `SysError`

On Error End

statement✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**`On Error End`**Revision:**

June, 2001 (Release 5)

Description:

There are two possible outcomes when using this statement and they depend on other factors in your program. If you have not established any other error handling routine, then you may use this routine to turn off all error checking. Errors such as file errors will be ignored. It will be your responsibility to track them manually after each file access statement by checking the function `Error`. This concept is demonstrated in the example below.

A second use involves programs where you have set up your own error handling routines. You may toggle between FB's error handling and your program's built-in error handlers by using `On Error End` to turn off FB's handlers and use the ones in your program. Alternatively, you may use `On Error Return` to reinstate FB's handlers.

Example:

```
// Manual, line-by-line error handling

Print "This program will produce a file error"
Print "that is completely ignored."

On Error End
Open "I",#1,"this file does not exist"

Print
Print "The error has occurred and was not flagged."

Print "The error number is"; Error And &FF
Print "In file number    "; Error >> 8
Print "The system error is"; SysError
```

Note:

If you turn off error checking (`On Error End`) and you get an error with `x = Error`, then your program must clear the error variable with `Error = _noErr`

See Also:

```
On Error Fn; On Error Return; Error; SysError
```

On Error Return

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

On Error Return

Description:

Use this statement to reinstate FB's standard error checking routines. After this statement is invoked, FB will display an error dialog and halt the program when a file error is encountered. If your program has established its own On Error Fn vector, it will be ignored. If you wish to return control to your internal routine or turn off FB's error checking, use On Error End.

See Also:

On Error Fn; On Error End; Error; SysError

On Event

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
On Event {Fn userFunction|Gosub{lineNumber|"stmtLabel"}}
```

Description:

This statement designates a particular function or subroutine as a system-event handling routine. A system-event handling routine is called in response to any event which the MacOS puts into the event queue designated for your program. This includes various kinds of low-level events such as mouseclicks and keypresses, as well as high-level events such as Apple Events. See the `Event` function for more information.

After a system event occurs, `FB^3` does not call your designated routine immediately. Instead, your program continues executing until a `HandleEvents` statement is reached. At that time, `HandleEvents` will call your designated routine once for each system event that occurred; your designated routine should examine the `Event` function to get information about the event.

If there are no events in the system queue when your program executes `HandleEvents`, `FB^3` calls your designated routine once, passing it a “null” event in the `Event` record.

Even if you don’t designate a system-event handling routine, `FB^3` often uses system events to determine whether other kinds of interesting events have occurred. For example, if the queue contains a system event of type `_mButDwnEvt` (indicating that the user has pressed the mouse button), `FB^3` checks whether the mouse was clicked inside a button, or in the menu bar, or in the “close box” of a window, etc., and may generate an event such as a dialog event or a menu event that your program can detect in other event handling routines.

By designating a system-event handling routine, your program can “intercept” events like `_mButDwnEvt`, before `FB^3` has a chance to interpret them and report them to your other event handling routines. (When a system event occurs, `FB^3` always calls your system-event handling routine first, before any of your other designated event handling routines.) This allows your program to customize the way it responds to system events, in case FutureBASIC’s “standard” responses don’t meet your needs. If you handle an event within your system-event handling routine, you can inhibit `FB^3` from further interpreting the event by setting the `_evtNum` field in the event record to `_nullEvt` before returning from your routine, as illustrated here:

```
Local Fn DoEvent
  evtPtr& = Event
  Select Case evtPtr&.evtNum%
    //[Handle the event as desired in here]
  End Select
  // "Hide" the event from further handling by FB:
  evtPtr&.evtNum% = _nullEvt
End Fn
```

Another good reason to designate a system-event handling routine is so that your program can respond to high-level events such as Apple Events. See the “Event Manager” chapter in *Inside Macintosh: Macintosh Toolbox Essentials*, for a discussion of high-level events.

Note:

If you use the `On Event Fn userFunction` syntax, then `userFunction` must refer to a function which was defined or prototyped at an earlier location in the source code. Your system-event handling function should not take any parameters, nor return a result.

See Also:

Event

On FinderInfo

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
On FinderInfo {Fn userFunction|Gosub{lineNumber| "stmtLabel"}}
```

Revision:

February, 2002 (Release 6)

Description:

You establish this vector before entering your event loop. When a file is dropped onto your application's icon or one of your applications files is double-clicked from the Finder, the specified routine is called with information on how to open the file.

The routines are set up to handle up to 1024 files at a time. If this number is insufficient, you will need to change the Dim statement in the header file named "Subs Files.Incl". There are three global values maintained for this vector.

gFBFndrInfoCount	The number of files pending in the queue.
gFBInfoSpec(1024) As FSSpec	An array of file spec records. There is one file spec record for each file that needs to be opened or printed.
gFBInfoAction%(1024)	A boolean value that is zero if the file is to be opened and non-zero if it is to be printed.

Example:

This routine establishes a function that is called when a file is dropped onto the compiled version of the application. It also shows how to determine if a file is to be printed or opened.

```

/*
    Build the application,
    then drop a text file On to it
*/
Local Fn MyOpenFile ( fs As ^FSSpec )
    Print "FileName: ";fs.name
End Fn

Local Fn MyFinderInfo
    Dim As FSSpec fs
    Dim As Short @ count, action, j
    Dim As OSType @ fType
    count = 0 // set To ask "How many?"
    action = FinderInfo( count, fs, fType ) // FSSpec & OSType
    Long If ( count > 0 ) // at least one file wants in
        For j = 1 To count // process them all
            count = -j
            // FSSpec & OSType
            action = FinderInfo( count, fs, fType )
            If ( action == _finderInfoOpen ) And ¬
                ( fType == _"TEXT" ) ¬
                Then Fn MyOpenFile( fs )
        Next
        Fn ClearFinderInfo // in Subs Common.Incl
    End If
End Fn

On FinderInfo Fn MyFinderInfo

Menu 1,0,1, "File"
Menu 1,1,1, "Quit/Q"
Window 1
Do
    HandleEvents
Until 0

```

See Also:

FinderInfo; **Open**; *Appendix A: Specifying Files And Directories*; *Appendix H: File Spec Records*; *Appendix I: Printing*; **Usr ScanFolder**; **Resources** statement

On <expr> Gosub

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
On expr Gosub {"label1" | lineNumber1} [, {"label2" | lineNumber2} ...]
```

Description:

This statement calls one of the subroutines indicated by a *label* or a *lineNum*, according to the value of *expr*. If *expr* equals 1, then the subroutine at *label1* or *lineNum1* is called; if *expr* equals 2, then the subroutine at *label2* or *lineNum2* is called, and so on. If *expr* is less than 1, or greater than the number of *label* 's and *lineNum* 's in the list, then the On <expr> Gosub statement does nothing.

See Also:

```
Gosub; On <expr> Goto; Select
```

On <expr> Goto statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
On expr Goto {"label1"|lineNum1}[,{"label2"|lineNum2}...]
```

Description:

This statement jumps to one of the indicated *label* 's or *lineNum* 's, according to the value of *expr*. If *expr* equals 1, then the program jumps to *label1* or *lineNum1*; if *expr* equals 2, then the program jumps to *label2* or *lineNum2*, and so on. If *expr* is less than 1, or greater than the number of *label* 's and *lineNum* 's in the list, then the On <expr> Goto statement does nothing.

See Also:

Goto; On <expr> Gosub; Select

On LPrint**statement**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
On LPrint {Fn userFunction|Gosub{lineNumber|"stmtLabel"}}
```

Description:

This statement designates a particular function or subroutine as an idler routine to be called by the printer driver during printing. The actual number of times that this function is called depends on the driver and the hardware.

You may use the `On LPrint` vector to rotate through a series of cursors, check for a Command-Period key press, or update the status of a print job.

Note:

If you use the `On LPrint Fn userFunction` syntax, then `userFunction` must refer to a function which was defined or prototyped at an earlier location in the source code.

See Also:

`HandleEvents`; `Menu` function

On Menu

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
On Menu {Fn userFunction|Gosub{lineNumber|"stmtLabel"}}
```

Description:

This statement designates a particular function or subroutine as a menu-event handling routine. A menu-event handling routine is called in response to the user selecting an item from a menu. This includes menu items that your program puts into menus on the menu bar, but it doesn't include items in pop-up menus; see the `Menu` function for more information.

When the user clicks on the menu bar, FB^3 does not open up the menu immediately. Instead, your program continues executing until a `HandleEvents` statement is reached. If the mouse button is still down at that time, `HandleEvents` then opens the menu, tracks the user's selection, then calls your menu-event handling routine if the user selected a menu item. Your routine should examine the `Menu(_menuID)` and `Menu(_itemID)` functions to get information about the event.

Note:

If you use the `On Menu Fn userFunction` syntax, then `userFunction` must refer to a function which was defined or prototyped at an earlier location in the source code. Your menu-event handling function should not take any parameters, nor return a result.

See Also:

`HandleEvents`; `Menu` function

On Mouse

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
On Mouse {Fn userFunction|Gosub{lineNumber|"stmtLabel"}}
```

Description:

This statement designates a particular function or subroutine as a mouse-event handling routine. A mouse-event handling routine is called in response to a mouseclick which occurs inside the content region of the currently active window (but not inside any buttons, scrollbars, edit fields nor picture fields).

After such a mouseclick occurs, FB^3 does not call your designated routine immediately. Instead, your program continues executing until a `HandleEvents` statement is reached. At that time, `HandleEvents` will call your designated routine once for each mouseclick event that occurred; your designated routine should examine the `Mouse <event>` functions to get information about the event.

Note:

If you use the `On Mouse Fn userFunction` syntax, then `userFunction` must refer to a function which was defined or prototyped at an earlier location in the source code. Your mouse-event handling function should not take any parameters, nor return a result.

If your program does not use `HandleEvents`, you can use the `Mouse(_down)`, `Mouse(_lastMVert)`, `Mouse(_lastMHorz)` and many other position functions (outlined in the FB Mouse Group of the constants document) to track mouse activity.

See Also:

```
Mouse <event>; Mouse(_down); Mouse <position>; HandleEvents
```

On Overflows

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
On Overflows {Fn userFunction | Gosub { lineNumber | "stmtLabel" } }
```

Description:

This statement designates a particular function or subroutine as an overflow handling routine. An overflow handling routine is called in response to any math operation involving a division by zero, or any floating-point math operation that results in a number larger than FB^3 can handle. If you don't designate any overflow handling routine, FB^3 will attempt to continue evaluating the expression, but the resulting number is unpredictable.

See Also:

Appendix C: *Data Types and Data Representation*

On Stop

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
On Stop {Fn userFunction|Gosub{lineNumber| "stmtLabel"}}
```

Description:

This statement designates a particular function or subroutine as a stop-event handling routine. A stop-event handling routine is called just before your program ends. This works whether your program ends as a result of a statement (such as `Stop` or `End`), or a ⌘-period keypress by the user, or because it has executed the last line in the program.

A stop-event handling routine is useful for allowing your program to do any last-minute “clean-up” work, such as saving files, before your program ends.

Note:

You can’t use a stop-event handling routine to prevent your program from stopping. Your program will end as soon as it returns from the routine. To prevent the user from stopping your program by means of a ⌘-period keypress, use the `On Break` statement.

See Also:

```
On Break; Stop; End; System
```


On Timer

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```

On Timer(interval) {Fn userFunction|Gosub{lineNumber|"stmtLabel"}}
Timer End
Timer Off

```

Revision:

January 2, 2001 (Release 4)

Description:

This statement designates a particular function or subroutine as a timer-event handling routine. A timer-event handling routine is called periodically according to a time interval that you specify.

Setting *interval* to a nonzero value causes timer events to be initiated, which causes FB^3 to periodically check whether it's time to call your routine. If *interval* is positive, it specifies the timer interval in seconds. If *interval* is negative, then `Abs(interval)` specifies the interval in ticks (a tick is approximately 1/60 second). Setting *interval* to zero does not initiate timer events; in this case, you can use the `Timer` statement to initiate timer events later in your program.

After timer events have been initiated, FB^3 checks its internal timer whenever a `HandleEvents` statement is executed. If FB^3 checks its timer and finds that at least *interval* seconds (or `Abs(interval)` ticks) have elapsed since the last time your designated routine was called, it calls your designated routine again.

Timer events are queued. If the timer interval is one second and your program is otherwise occupied for a period of 10 seconds, you will receive 10 timer events as soon as the program becomes idle. You may flush this queue by using `Timer Off` or `Timer End`.

Designating a timer-event routine is useful when your program needs to perform periodic activities at particular intervals. To ensure that your timer-event handling routine is called at the right moments, your program should execute `HandleEvents` as often as possible.

Note:

You can use the `Timer` statement to change the timing interval. You cannot disable timer events once they've been initiated.

See Also:

`Timer statement`; `HandleEvents`

Open

statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Open "method[Fork]",fileID,path$|@FSSpec[, [recLen] [,refNum%[,dirID&]]]
```

Revision:

February, 2002 (Release 6)

Description:

This statement opens a file so that you can read from it and/or write to it. If you specify a *method* other than "I", the Open statement also creates the file if it doesn't already exist.

The parameters are interpreted as follows:

- *method*

Specify one of the following letters:

I	Open for input (reading) only. The file must already exist. Other processes may read from the file (but not write to it) while it's open with this method.
O	Open for output (writing) only. If the file already exists, all of its current contents will be destroyed. You have exclusive access to the file (no other process can read from it nor write to it) while it's open with the "O" method.
R	Open for "random access." You can either read from or write to the file. The "file mark" (which indicates where the next read or write operation will occur) is placed initially at the beginning of the file. If you write to the file, you only replace those bytes which you're writing; the rest of the file's contents are unaffected. You have exclusive access to the file.
A	Open for "append." This is just like method "R", except the file mark is placed initially at the end of the file. This method is normally used when you want to add data to the end of an existing file.
N	Open for non-exclusive random access. This is just like method "R", except that other processes may read from the file while you have it open. Your process is the only one allowed to write to the file.

- *Fork*

Specify one of the following letters:

D	Open the "data fork." This is the default value.
R	Open the "resource fork." You should only specify this value if you're doing something like copying an entire resource fork from one file to another, or examining the internal structure of the resource fork. You should not use the Open statement when you want to read or write individual resources; in that case, use a Toolbox function such as HOpenResFile instead. Do not use the Open statement to create a new resource fork; use a Toolbox function such as HCreateResFile instead.

- *fileID*

Specify a number in the range 1 through 255 which is not being used by any other currently open file. You can use this number to identify the open file in statements and functions such as `Read#`, `Write#`, `Eof`, `Lof`, etc. The *fileID* number is associated with the file until you close the file.

- *path\$*, *refNum%*, *dirID&*

These three values indicate the name and location of the file to be opened. See Appendix A: *Specifying Files and Directories*, to learn how these parameters are used.

or...

- *@FSSpec*

A file spec record may be substituted for the path name. If such a record is used, the *refNum%* and *dirID&* parameters are not used. (They are already part of the file spec record.) See the new `Files$` parameters for information about obtaining a file spec record. See Appendix H for information about file spec records.

- *recLen*

This value indicates the length of the records in the file; naturally, it's most useful when the file consists of fixed-length records. The value you specify is used when you execute statement and functions such as `Record`, `Rec`, `Loc` and `Lof`. If you omit this parameter, a default value of 256 is used. If the file doesn't consist of fixed-length records, it's often most convenient to set *recLen* to 1.

See Also:

`Close`; `InKey$`; `Input#`; `Print#`; `Read#`; `Read File`; `Read Field`; `Write#`;
`Write File`; `Write Field`; `Record`; `Rec`; `Loc`; `Lof`; `Eof`; `Usr _fileAddr`;
`Files$`; `ParentID`

Open "C"

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Open "C",portID,baud [, [parity] [, [sTopbit] [, [wordLength] [, buffer]]]]
```

Revision:

March 2, 2001 (Release 5)

Description:

This statement opens a serial communications port (the modem port or the printer port) so that your program can write to or read from a serial device. The optimal values for the various parameters depend on the device and the desired communications protocol; see the device's manual for more information. The parameters are interpreted as follows:

- portID*** Set this either to `_modemPort` or to `_printerPort` or to any port specified to a maximum -8. (Ports are numbered from -1 for the printer port to -8.) The `_modemPort` value also usually works to communicate with a built-in modem. Some Macintosh computers provide different values. A powerbook generally uses `_modemPort` as the infra red port. and `_printerPort` as the internal modem. USB adapters such as the Keyspan adapter will provide different values if the device is connected before booting as opposed to plugging it in after the computer is running.
- baud*** Set this to one of the following values: 110; 300; 1200; 1800; 2400; 3600; 4800; 7200; 9600; 19200; 38400; 57600, 115200, 230400.
- parity*** Set this to one of the following values: `_noParity`; `_oddParity`; `_evenParity`. The default value is `_noParity`.
- sTopbit*** Set this to one of the following values: `_oneSTopBit`; `_twoSTopBits`; `_halfSTopBit` (1.5 stop bits). The default value is `_oneSTopBit`.
- wordLength*** Set this to one of the following values: `_fiveBits`; `_sixBits`; `_sevenBits`; `_eightBits`. The default value is `_sevenBits`. Note: do not set this parameter to the values 5, 6, 7 or 8: these are different from the values of the symbolic constants.
- buffer*** Set this to a number in the range 1 through 32,768. This parameter indicates how many bytes to allocate for an input buffer. The input buffer stores data that is being received, even when the program is not reading it, allowing the program to process data while data is being received in the background. The default value for *buffer* is 4096 bytes. To determine the number of unread characters currently in the buffer, use `Lof (portID,1)` .

Reading Data

To read incoming data from an open serial port, use the same commands that you would use to read data from a file; e.g., `Input#`, `Read#`, etc. Since it's difficult to predict when (if ever) the data will come in, it's best to design your program so that it won't get "stuck" on a single statement waiting for incoming data. Instead, you should execute a loop that periodically checks whether there is any data to read. This will allow your program to proceed with other activities while it's waiting; or to quit waiting if too much time has elapsed.

There are basically three ways to check whether there is any data available in the buffer:

- You can check the value of `Lof(portID,1)`. This will return zero if no data is available to read; otherwise, it returns the number of bytes waiting to be read.
- You can use the `Read# portID,stringVar$;0` statement. By specifying `“;0”`, you instruct the `Read#` statement to return immediately if there is no data available; if there is data, the statement reads all the characters currently in the input buffer (up to the maximum allowable length of `stringVar$`), and puts them into `stringVar$`. You can use `Len(stringVar$)` after the `Read#` statement to determine how many (if any) characters were read.
- You can use the `InKey$(portID)` function. It will either return one character from the buffer, or a null string if the buffer was empty.

Writing Data

To write data out to an open serial port, use the same commands that you would use to write data to a file; e.g., `Print#`, `Write#`, etc.

FB Runtime Globals

FB^3 has several reserved global variables. (See `Subs Files.Incl` in Header folder)

```
gFBHasComTB%           //true if comm Toolbox is used...
gFBSerialPortCount%    //number of com port
gFBSerialName$(n)      //serial port name
gFBSerialInName$(n)    //input buffer name
gFBSerialOutName$(n)   //output buffer name
gOSXSerialInitd        //≠0 if serial initd under OS X
```

After any communications port has been opened or after you make your own call to the runtime `Fn FBInitSerialPorts`, you may refer to `gFBSerialPortCount%` for the total number of devices (maximum 8). `gFBSerialOutName$(n)` contains the name of the device. With this in mind, the serial ports can best be referred to by name rather than number when multiple ports are present or when USB devices are in use for the purpose of emulating serial ports.

To search all available communication ports use the folling lines. This is especially important if a USB/serial port adapter is inserted after the program has started.

```
gFBSerialportCount% = 0
//      This is For OS-X
Long If System(_sysVers) => 1000
    gOSXSerialInitd = _false
End If
```

Example:



CD Example: CTB demo

See Also:

Close; Handshake; Loc; Lof; Input#; Read#; Read File; Read Field;
InKey\$; Print#; Write#; Write File; Write Field

Open "UNIX"

statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Open "UNIX", fileID, UNIXcommand$
```

Revision:

August, 2002 (Release 7)

Description:

Mac OS X brings a wealth of commands with its UNIX foundation. These commands are easy to access from FutureBASIC. You may establish one or several simultaneous channels by opening them as files. Each channel should have a unique file number. There is a theoretical maximum of 255 files or channels that may be opened at once.

Once the channel has been opened, you read from the file to accept the resulting list from UNIX. (Exception: some UNIX commands perform a task without responding to the user, so no file reading is required.) When the operation is complete, close the channel as you would any file; with the `Close` statement.

There are literally hundreds of books written about UNIX commands. Suffice it to say that if the Mac supports a specific UNIX command, it may be accessed through `Open "UNIX"`

Example:

```

Dim As Str255 a
Window 1,, (0,0)-(600,550)
Text _monaco, 10
// Print a title-string then list a directory
Open "UNIX", 2, "echo ""Root DirectTory""; ls -l"
Do
    Line Input #2, a
    Print a
Until Eof(2)
Close 2

Do
    HandleEvents
Until 0

```

Note:

`Open "UNIX"` works only in Mac OS X.

See Also:

`Line Input`; `Close`

Or**operator**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
result& = exprA {Or | ||} exprB
```

Description:

Expression *exprA* and expression *exprB* are each interpreted as 32-bit integer quantities. The `Or` operator performs a “bitwise comparison” of each bit in *exprA* with the bit in the corresponding position in *exprB*. The result is another 32-bit quantity; each bit in the result is determined as follows:

<i>Bit value in exprA</i>	<i>Bit value in exprB</i>	<i>Bit value in resultat&</i>
0	0	0
1	0	1
0	1	1
1	1	1

The `Or` operator can also be used to join two “condition clauses” for use in statements like `If`, `While` and `Until`. For example:

```
If n > 17 Or myName$ = "Smith" Then Beep
```

This statement produces a beep if either `n > 17` is true, or `myName$ = "Smith"` is true, or both.

Even when it’s used to join condition clauses, the `Or` operator still does a “bitwise comparison.” This happens because FB^3 actually assigns a numeric value to every condition clause, depending on whether the clause is true or false. For example, the clause `n > 17` is evaluated as `-1` if it’s true, or as `0` if it’s false. Conversely, any numeric expression is judged as “true” if it’s non-zero, or as “false” if it’s zero.

Example:

In the following example, expressions are evaluated as true or false before a decision is made for branching. The logical expression `state$ = "IL"` is true, and therefore evaluated as `-1`. The expression `state$ = "CA"` is false, and is therefore evaluated as `0`. Then the bitwise comparison `(-1) Or (0)` is performed, resulting in `-1`. Finally, the `Long If` statement interprets this `-1` result as meaning “true,” and therefore executes the first `Print` statement.

```
state$ = "IL"
Long If state$ = "IL" Or state$ = "CA"
    Print "Okay"
Xelse
    Print "Invalid state"
End If
```

The example below shows how bits are manipulated with `OR`:

```
DefStr Long
Print Bin$(923)
Print Bin$(123)
Print "-----"
Print Bin$(923 Or 123)
```

Program output:

```
0000000000000000000000000000000000001110011011  
0000000000000000000000000000000000001111011  
-----  
0000000000000000000000000000000000001111111011
```

See Also:

Not; And; Xor; Appendix D: *Numeric Expressions*

Output statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Output [File] "filePath"
```

Description:

Use this statement to specify a name for the application file that's created when you "Build" a project. If your program doesn't contain an `Output` statement, then FB^3 will use a default name (as set in your preferences) when you select "Build" (⌘B from the "Command" Menu).

The *filePath* can be:

- A simple file name. The application file is saved in your project folder.
- A relative path name (starting with a colon). The path is relative to your project folder.
- A full path name. The application file is saved in the specified folder.

The `Output` statement can appear anywhere in your program, but most commonly it appears somewhere near the beginning.

Note:

`Output` is a non-executable statement, so you can't affect its operation by putting it inside conditional execution structures like `Long If...End If`. You can, however, conditionally include or exclude it using `Compile Long If`.

See Also:

`Compile`; `Compile Long If`; `Resources`

Override

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

Override Local Fn fnName
Override Runtime fnName
Override EnterProc fnName
Override _constantName = newValue

```

Revision:

May 5, 2000 (Release 3)

Description:

When used to redefine a function, Runtime, or EnterProc, the `Override` command instructs the compiler to begin using a new address for a specific routine. The new version must contain the same parameters as the older version. If no older version exists the function is created.

The following example overrides FB's `Print` statement by putting print marks around everything that is printed.

```

Override Runtime PrintString
  call DrawString("")
  Fn FBPrintString // DrawString(gFBStr&.FBStrAcc$)
  call DrawString("")
End Fn

```

`Override` may also be used to change the value of a constant.

Note:

When you `Override` a constant, any code compiled after the override is affected. Constants are not variables. They are only examined at compile time. It is therefore not possible to override a constant that is used in the runtime since FB^3 has already compiled the entire runtime before your `Override` is ever encountered.

Page**function**

*✓ Appearance**✓ Standard**✓ Console*

Syntax:

```
lineCount = Page
```

Description:

This function returns a count of the text lines which have been sent to the printer on the current page. Its value is incremented each time a carriage-return character is sent to the printer, and is reset to zero each time a printed page is ejected.

See Also:

```
CsrLin; Close LPrint; Clear LPrint; Route
```

Page	statement		
	✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>

Syntax:

Page

Description:

This statement is identical to the `Clear LPrint` statement.

See Also:

`Clear LPrint`

Page LPrint

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

Page LPrint

Description:

This statement prints the content area of the current output window to the selected printer. The picture is a bit mapped copy of screen pixels. Before the printing starts, the user is offered a page setup dialog.

See Also:

Def LPrint; Route

ParentID statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
ParentID = dirID&
```

Description:

This statement assigns a directory ID number to be used as a default *dirID&* parameter in the next `Open`, `Kill` or `Rename` statement, in case that statement does not explicitly specify a *dirID&* parameter. After the next `Open`, `Kill` or `Rename` statement is executed, the `ParentID` value is implicitly reset to zero.

The `ParentID` value is provided for compatibility with older versions of FutureBASIC. It's generally preferable to specify the *dirID&* value as an explicit parameter in `Open`, `Kill` and `Rename`.

Note:

The `ParentID` statement does not change your process' "default directory." To do that, use the `Folder` function, or the Toolbox functions `SetVoL` or `HSetVol`.

See Also:

`Open`; `Kill`; `Rename`; [Appendix A: Specifying Files And Directories](#)

Peek

functions

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

byteValue`      = Peek [Byte] (address&)
shortIntValue% = Peek Word (address&)
longValue&      = Peek Long (address&)

```

Shorthand syntax:

```

byteValue`      = |address&|
shortIntValue% = {address&}
longValue&      = [address&]

```

Description:

The `Peek` functions look at the 1, 2 or 4 bytes of data which begin at `address&`, and return them as a byte integer, short integer or long integer value, respectively. The `address&` should be a long integer expression, or a `Pointer` or `Handle` variable.

The value returned by a `Peek` function will be interpreted either as a signed or unsigned value, depending on what type of variable it's assigned to. If the value is not assigned to any variable, it's usually interpreted as a signed value.

See Also:

`Poke`; `VarPtr`

Pen statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Pen [penWidth][,[penHeight][,[visible][,[mode][,pattern]]]]
```

Description:

This statement alters the characteristics of the drawing “pen” in the current output window. The pen characteristics affect the appearance of QuickDraw shapes (lines, ovals, rectangles, etc.) that are subsequently drawn in the window. If you omit any parameter, the corresponding characteristic is not altered. The parameters are interpreted as follows:

- *penHeight* and *penWidth*

These specify the height and width of the pen in pixels. They must be positive integers. Taller, wider pen sizes produce thicker lines and borders.

- *visible*

If you set this to `_false`, subsequent drawing won’t be visible on the screen (but it will still be “recorded,” if you have turned on picture recording (see the `Picture On` statement)). If you set *visible* to `_true`, subsequent drawing will be visible.

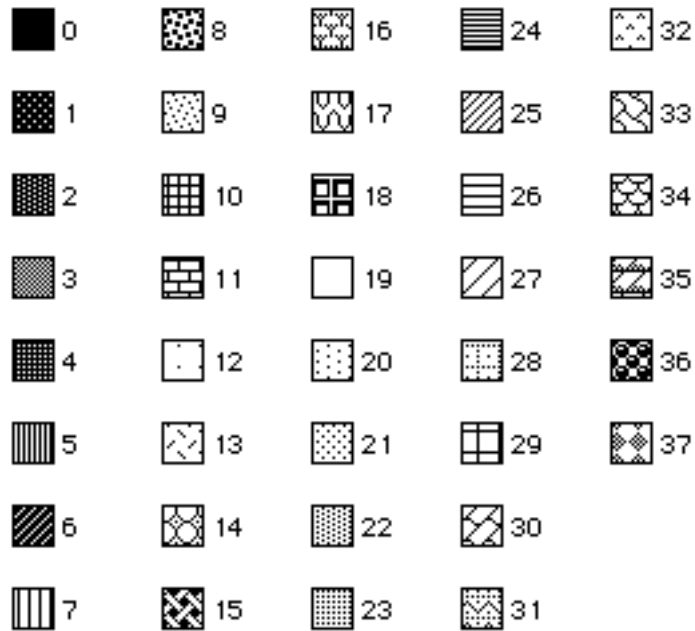
- *mode*

This determines how the pen behaves when you draw over existing images in the window. You can specify any of the pattern transfer modes described in Inside Macintosh: *Imaging With QuickDraw*. Usually you will use one of the following constants:

<code>_patCopy</code>	<code>_transparent</code>
<code>_patOr</code>	<code>_addOver</code>
<code>_patXor</code>	<code>_addPin</code>
<code>_patBic</code>	<code>_subPin</code>
<code>_notPatCopy</code>	<code>_adMax</code>
<code>_notPatOr</code>	<code>_subOver</code>
<code>_notPatXor</code>	<code>_adMin</code>
<code>_notPatBic</code>	<code>_blend</code>

- pattern*

This determines the pattern that will be used to draw lines, and to frame or fill shapes. Specify a number in the range 0 through 37 to get one of the following system patterns:



Console behavior:

When you use the Console runtime, Pen switches to the Graphics Window before executing.

Note:

To change the pen's color, use the `Color` or `Long Color` statement. To change the appearance of text, use the `Text` statement.

See Also:

`Plot`; `Box`; `Circle`; `Fill`; `Color`; `Long Color`; `Text`

Picture**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
pictureHandle& = Picture
```

Description:

This function returns a handle to the picture recorded with the most recent pair of `Picture On`/`Picture Off` statements. This handle is the same handle returned by:

```
Picture Off, pictureHandle&
```

You can specify this handle in the `Picture` statement when you want to draw the picture. You can also pass the picture handle to any of a number of Toolbox routines which require a picture handle as a parameter.

Note:

Your program is responsible for releasing the memory occupied by pictures created with the `Picture On`/`Picture Off` statements. You should normally use the `Kill Picture` statement to do this, once you're finished using the picture handle. However, if you turn the picture into a resource (using the Toolbox routine `AddResource` or PG's `Fn pGreplaceRes`) then you should not dispose of the picture.

See Also:

```
Picture On/Off; Picture statement
```

Picture

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Picture [ (h1,v1) ] [ -(h2,v2) ] [ ,pictureHandle& ]
```

Description:

This function draws a picture in the current output window, or to the printer if output is currently routed to the printer. If you specify *pictureHandle&*, the picture referenced in that handle is drawn; this can be a handle returned by the `Picture` function, or a `PICT` resource handle, or any other valid picture handle. If you don't specify *pictureHandle&*, the picture which was recorded by the most recent pair of `Picture On`/`Picture Off` statements is drawn.

The $(h1, v1)$ and $(h2, v2)$ parameters specify the upper-left and lower-right corners of a rectangle. If you specify both $(h1, v1)$ and $(h2, v2)$, the picture is scaled to fit the indicated rectangle. If you specify only $(h2, v2)$, the picture is scaled to fit the rectangle $(0, 0) - (h2, v2)$. If you specify only $(h1, v1)$, the picture is drawn unscaled, with its upper-left corner at $(h1, v1)$. If you specify neither corner, the picture is drawn unscaled, using the same frame rectangle that was used to record the picture.

See Also:

`Picture On/Off`; `Picture` function

Picture Field

statement

✓ *Appearance*

✓ *Standard*

✗ *Console*

Syntax:

```
Picture Field [#]pfID [, [pict] [, [rect] [, [type] [, [just]]]]
```

Revision:

February, 2002 (Release 6)

Description:

Use this statement to create a new picture field in the current output window, or to modify the characteristics of an existing picture field. A picture field displays a picture which is automatically refreshed when necessary, and which can optionally respond to mouse clicks.

There are important differences in how picture fields are handled in the older Standard BASIC versus the newer Appearance Runtime. In Standard BASIC, picture fields were actually edit fields that had a destination rectangle with no height. (The top of the destination rectangle was equal to the bottom.) The contents of those edit fields consisted of data that told FB what to display. For instance, if the contents were. "%1001" then FB knew to display a resource picture with an ID of 1001.

Appearance picture fields are buttons and as such inherit the full power of the Appearance Manager's ability to display and manage them properly. There is, however, a difference between a picture *field* and a picture *button*. A picture *field* is fully managed as a user pane (a special type of button) by the Appearance Runtime. In other words, FB tells the toolbox that it should call back into the runtime for the details of managing the picture. A picture *button* is a standard CDEF (control definition) that is managed by the Appearance Manager. Both are buttons. But Picture Fields are generally display items where Appearance Buttons that employ pictures are generally action items.

Appearance Manager picture fields abandon the more obscure framing conventions that were available in the Standard BASIC runtime. For instance, frame types such as `_round`, `_rounder`, and `_roundest` are no longer available.

The parameters are interpreted as follows.

pfID

This is the ID number of the picture field. To create a new picture field, specify a *pfID* value which is different from the ID of any existing picture field, and any existing edit field, in the current window. When you're creating a new picture field, the *pict* and *rect* parameters are also required. To modify an existing picture field, specify the ID number of the existing field, and any other parameters that you want to modify; any parameter that you omit won't have its corresponding characteristic changed.

pic

This parameter specifies the picture to display in the picture field. You can specify it in any of the following forms:

- `resName$` The name of a `PICT` resource in a currently open resource file.
- `%resID%` A “%” symbol followed by the resource ID number of a `PICT` resource in a currently open resource file.
- `&picHandle&` An “&” symbol followed by a picture handle.

If you use a `PICT` resource, it’s best to use one that’s purgeable so that the memory it occupies will be properly released (this memory is not automatically released when the picture field is closed). Don’t close the resource file containing the picture until after you close the picture field or you put a different picture into the field. If you use a picture handle, make sure you do not release its memory (don’t call `Kill Picture`) until after you close the picture field or you put a different picture into the field.

rect

This parameter specifies the rectangle in which the picture will appear; no part of the picture will be drawn outside of this rectangle. You can specify it in either of the following forms:

- `(x1%,y1%)-(x2%,y2%)` These coordinates specify two diagonally opposite corners of the rectangle.
- `rectAddr&` A pointer or long integer expression. This is interpreted as the address of a standard 8-byte `Rect` structure.

type

This is an integer which specifies several characteristics about the field.

“Clickable” types.

Picture fields of these types will generate an `_efClick` Dialog event (Standard BASIC) when the user clicks in them.

<i>type</i>	<i>Value</i>	<i>Description</i>
<code>_pfFramed</code> (<i>Appearance Manager</i>)	0	A frame is drawn around the field
<code>_pfNoFramed</code> (<i>Appearance Manager</i>)	256	No frame is drawn around the field
<code>_pfClickable</code> (<i>Appearance Manager</i>)	512	A dialog event (<code>_pfClick</code>) is received if the picture is clicked.
<code>_pfInvertsOnClick</code> (<i>Appearance Manager</i>)	1024	If the picture is clickable (<code>_pfClickable</code>) it will be inverted when clicked.
<code>_pfTransparent</code> (<i>Appearance Manager</i>)	2048	White parts of the picture allow the background to show through.
<code>_framed</code>	2	A frame is drawn around the field. The picture is “click-reversed” while the mouse button is held down in it, which means the image’s colors are inverted.
<code>_framedNoCR</code>	1	A frame is drawn around the field. The picture is not “click-reversed” when the user clicks in it. This is the default type.
<code>_noFramed</code>	4	Like <code>_framed</code> , but no frame is drawn.
<code>_noFramedNoCR</code>	3	Like <code>_framedNoCR</code> , but no frame is drawn.

“Non-clickable” types.

Picture fields of these types will not generate any Dialog event when the user clicks in them.

<i>type</i>	<i>Value</i>	<i>Description</i>
<code>_statFramed</code>	5	Frame is drawn.
<code>_statNoFramed</code>	7	No frame is drawn.
<code>_statFramedGray</code>	9	Frame is drawn; frame and picture are dimmed.
<code>_statNoFramedGray</code>	11	No frame is drawn; picture is dimmed.
<code>_statFramedInvert</code>	13	Frame is drawn; picture’s colors are inverted.
<code>_statNoFramedInvert</code>	15	No frame is drawn; picture’s colors are inverted.
<code>_statFramedInvert</code> + <code>_hilite</code>	29	Frame is drawn; picture is highlighted using system highlight colors.
<code>_statNoFramedInvert</code> + <code>_hilite</code>	31	No frame is drawn; picture is highlighted using system highlight colors

You can also add any combination of the following constants to the `type` parameter to achieve different effects in appearance:

<i>type</i>	<i>Value</i>	<i>Description</i>
<code>_round</code> <code>_rounder</code> <code>_roundest</code> (= <code>_round</code> + <code>_rounder</code>)	32 64 96	<code>_round</code> causes the frame (if any) to be oval-shaped, and the picture to be clipped to the oval. <code>_rounder</code> and <code>_roundest</code> produce different rounded-rectangle shapes.
<code>_boldBox</code>	128	For framed picture fields, draws a thicker frame.

just

This parameter specifies how the picture will be scaled or cropped. Specify one of the following values:

<i>type</i>	<i>Value</i>	<i>Description</i>
<code>_scaledPict</code>	1	The picture is scaled to the size of the specified rectangle. This is the default value.
<code>_centerPict</code>	2	The picture is centered in the specified rectangle without scaling. If the picture is too big to fit, the top, bottom and/or sides of the picture will be clipped out.
<code>_cropPict</code>	3	The picture is drawn without scaling, with its upper-left corner corresponding to the upper-left corner of the specified rectangle. If the picture is too big to fit, the bottom and/or right side of the picture will be clipped out.

Note:

To remove a picture field from the window, use the `Edit Field Close` statement.

When you create, modify or click on a “clickable” picture field, all edit fields in the current window become inactive, and the picture field becomes “active.” When a picture field is “active,” you can retrieve various kinds of information about it using the `Window` function.

See Also:

`Edit Field`; `Edit Field Close`; `Picture On/Off`; `Window` function

Picture On/Off

statements

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Picture On [ (h1,v1) ] [ -(h2,v2) ]
Picture Off [,pictHandleVar&]
```

Description:

The `Picture On` statement initiates “picture recording” for the current output window. The `Picture Off` statement turns off picture recording.

While picture recording is on, all drawing commands and text-display commands which are sent to the window are “recorded” in a special data structure called a picture. The internal format of a picture is identical to that of a resource of type “`PICT`”. After picture recording is turned off, you can display or print the picture, or attach it to a picture field, or save it to disk (as a “`PICT`” resource or a “`PICT`” file).

Every picture has a frame, an imaginary rectangle which is stored as part of the picture’s data structure. Usually, but not always, the picture itself is contained within the frame. The frame is used as a reference to determine how the picture should be scaled and positioned when the picture is later displayed or printed. The picture recording is cropped to the recording window’s clip region, which may or may not be the same as the frame rectangle.

In the `Picture On` statement, the $(h1,v1)$ and $(h2,v2)$ parameters specify the upper-left and lower-right corners of a rectangle. If you specify both $(h1,v1)$ and $(h2,v2)$, they define the picture’s frame. Otherwise, the frame is determined as follows:

- If you specify only $(h1,v1)$, it becomes the frame’s upper-left corner; the frame’s lower-right corner is set to the current lower-right corner of the window.
- If you specify only $(h2,v2)$, the picture’s frame is set to the rectangle $(0,0)-(h2,v2)$.
- If you specify neither $(h1,v1)$ nor $(h2,v2)$, the picture’s frame is set to the window’s current rectangle.

In the `Picture Off` statement, a handle to the recorded picture is returned in `pictHandleVar&`, if it’s specified. `pictHandleVar&` must be a long-integer variable, or a `Handle` variable. The value returned in `pictHandleVar&` is the same as the value returned by the `Picture` function.

By default, drawing commands are not visible on the screen while they're being recorded. If you want the picture to be visible while you're recording it, you must call the Toolbox procedure `ShowPen` after you start recording. If you do this, `ShowPen` must be "balanced" by a call to the `HidePen` procedure, before you turn off picture recording. For example:

```

Picture On
Call ShowPen 'show while recording
' [execute drawing commands here]
Call HidePen 'balance the call to ShowPen
Picture Off

```

Only one window can have picture-recording enabled at any given time; you cannot "nest" `Picture On`/`Picture Off` pairs. If you switch output windows while picture recording is on, any drawing commands sent to the new output window will not be recorded.

You cannot "temporarily" turn off the recording of a picture. Each call to `Picture Off` completes a picture, and each call to `Picture On` starts a brand new picture. However, it is possible to effectively "append" one picture to another, by "inserting" an old picture inside a new one. For example:

```

Picture On
Circle 50,50,45
Picture Off, circlePict&
:
' "Append" more drawing commands:
Picture On                'start a new picture
Picture ,circlePict&      'This gets recorded in new pict
Box 20,20 To 50,50
:
Picture Off, pictHandle&  'contains circle & box

```

Note:

Your program is responsible for releasing the memory occupied by pictures created with the `Picture On`/`Picture Off` statements. Use the `Kill Picture` statement to do this, once you're finished using the picture handle.

Drawing commands which plot icons are not recorded in pictures.

See Also:

`Picture function`; `Picture statement`; `Kill Picture`; `Picture Field`

Plot**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

Plot h, v
Plot To h, v
Plot h1, v1 To h2, v2 [To h3, v3 ...]

```

Description:

This function draws a “point,” a line, or a series of connected lines, in the current output window, using the current pen size, pen mode, pen pattern and foreground color.

If you use the first syntax, a single “point” is drawn. This will actually be a little rectangular block whose dimensions are the same as the pen’s width and height, with its upper-left corner at point (h, v) .

If you use the second syntax, a line is drawn, having one endpoint at (h, v) . The line’s other endpoint will be one of the following:

- The last point specified in the most recent `Plot` statement (in any window);
- The (h, v) coordinates of the most recent `Box` statement (in any window) that actually specified the h and v parameters;
- $(0,0)$, if no `Plot` statement has yet been executed, and no prior `Box` statement that specified h and v has yet been executed.

If you use the third syntax, a line or a series of connected lines is drawn, with endpoints at the specified points.

Console behavior:

When you use the Console runtime, `Plot` switches to the Graphics Window before executing. You can’t use `Plot` to draw a line in the Text Window, or on the printer; use the Toolbox procedures `MoveTo` and `LineTo` instead. Alternatively, you can activate the graphics window and select Print from the File menu.

See Also:

`Pen`; `Box`; `Coordinate`; `Color`; `Long Color`

Poke

statements

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

Poke [Byte] address&, byteExpr
Poke Word address&, shortIntExpr
Poke Long address&, longIntExpr

```

Shorthand syntax:

```

| address&, byteExpr
% address&, shortIntExpr
& address&, longIntExpr

```

Description:

These statements copy the value in *byteExpr*, *shortIntExpr* or *longIntExpr* into the 1, 2 or 4 memory bytes (respectively) which start at location *address&*. The *address&* should be a long integer expression, or a `Pointer` variable.

See Also:

`Peek`; `VarPtr`; `BlockMove`; `Let`

Pop

statements

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Pop (var)
Pop {Byte|Word|Long} (address&)
```

Description:

These statements retrieve 1, 2 or 4 bytes off of the CPU stack, and adjust the stack pointer by a corresponding amount. `Pop` is the opposite of `Push`.

If you use the first syntax, the data retrieved from the stack is put into `var`, which must be a variable. The number of bytes retrieved from the stack depends on the data type of `var`:

<i>var type</i>	<i>Bytes retrieved from stack</i>
byte (signed or unsigned)	1
short integer/Word (signed or unsigned)	2
long integer (signed or unsigned); Pointer; Handle	4

(No other variable types are valid with `Pop (var)`.)

If you use the second syntax, the data retrieved from the stack is copied into the memory which begins at `address&`, which must be a long integer expression or a `Pointer` variable. The number of bytes retrieved from the stack depends on which keyword you use:

<i>Keyword</i>	<i>Bytes retrieved from stack</i>
Byte	1
Word	2
Long	4

Note:

In CPU68k compiles, `Pop` always adjusts the stack pointer by an even number of bytes. If you use `Pop (byteVar)` or `Pop Byte (address&)` in a CPU68k compile, the stack pointer will be adjusted by 2 bytes, even though only one byte is copied into `byteVar` or `address&`.

`Pop` is meant for careful use by advanced programmers. Your system can crash if the stack pointer is not adjusted carefully.

See Also:

`Push`

Pos

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
numCharacters = Pos(deviceType)
```

Description:

This function returns a number which means different things depending on the value of *deviceType*. Use one of the following values for *deviceType*.

- `_anyDev` `Pos(_anyDev)` returns information about the number of characters sent by the `Print` statement to screen windows or to the device specified by the most recent `Route` statement. The value of `Pos(_anyDev)` is incremented whenever you print a character (other than a carriage-return) to any open window or to the `Route`'d device. The value of `Pos(_anyDev)` is reset to zero whenever any of the following happens:

 - You send a carriage-return character to any window or `Route`'d device (this is usually the final character sent by a `Print` statement); or:
 - The text in any window or `Route`'d device reaches the right margin and wraps around to the next line; or:
 - You open a new window with the `Window` statement; or:
 - You start a new print job with the `Route _toPrinter` statement.

Note that `Pos(_anyDev)` often, but not always, represents the number of characters on the current line of text. However, because `FB^3` does not maintain separate `Pos` values for separate windows, the value returned by `Pos(_anyDev)` may represent the characters on a line in the current window, or on a line in a different window, or even the sum of the characters on lines in multiple windows.
- `_printerDev` `Pos(_printerDev)` returns the number of characters printed so far on the current line of text sent to the printer. The value of `Pos(_printerDev)` is incremented whenever you send a character (other than carriage-return) to the printer; the value is reset to zero whenever you send a carriage-return character to the printer, or the text reaches the right margin and wraps around to the next line.
- `_diskDev` `Pos(_diskDev)` returns information about characters sent to open files. The value of `Pos(_diskDev)` is incremented whenever you send a character (other than carriage-return) to any open file; the value is reset to zero whenever you send a carriage-return character to any open file. Note that if you have more than one file open, the value returned by `Pos(_diskDev)` reflects the sum of the characters sent to all the files you're writing to. If you write a total of more than 32767 characters (to all open files) without writing a carriage-return character, the number returned by `Pos(_diskDev)` is invalid.

Note:

To determine the current horizontal pen position (in pixels), use the `Window(_penH)` function.

See Also:

`CsrLin`; `Width`; `Page` function; `Window` function

PrCancel

function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
userCancelled = PrCancel
```

Description:

You should examine the value of `PrCancel` after executing the `Def LPrint` statement. `PrCancel` returns `_zTrue` if the user pressed the “Cancel” button in the Print Job dialog; or `_false` if the user pressed the “OK” button. If `PrCancel` returns `_zTrue`, your program should not continue with the print operation.

You can also call `PrCancel` after executing `Def Page`, to determine whether the user cancelled the Style dialog (the “Page Setup” dialog). However, your program normally does not need to take any special action in this case.

See Also:

```
Def LPrint; Def Page; PrHandle; Route
```

PrHandle

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**`pRecH& = PrHandle`**Revision:**

February, 2002 (Release 6)

Description:

This function returns a handle to the print record. The print record (also called a “TPrint record”) contains useful information about the printer and about the current print job (if any). For a complete description of the contents of the print record, see the “Printing Manager” chapter of Inside Macintosh: *Imaging with QuickDraw*. A few of the more useful fields from this record are listed here:

- `pRecH&..prInfo.rPage`

This gives the print page rectangle. You can get the width and height of the page as follows:

```
pageWidth = pRecH&..prInfo.rPage.right% - pRecH&..prInfo.rPage.left%
pageHeight = pRecH&..prInfo.rPage.botTom% - pRecH&..prInfo.rPage.Top%
```

Use these numbers to determine how much room is available for text and graphics. The page rectangle is affected by the user’s selections in the “Page Setup” dialog: for example, if the user selects “landscape” mode, the page width will be greater than the page height.

- `pRecH&..prJob.iFstPage%; pRecH&..prJob.iLstPage%`

These numbers indicate the first and last pages which the user wants to print. Your program should not print any pages outside of this range.

- `pRecH&..prJob.iCopies%`

The number of copies to print. This is the number of times your program should send the selected pages to the printer. Note that many newer printer drivers will always set this number to 1; such printer drivers will handle multiple copies internally.

Note:

`PrHandle` is partially emulated for Carbon. FB creates a 120 byte handle and fills in a few standard rectangles and other values so that programs won't be so likely to break during the transition period to OS-X. But these emulations are not something that you should depend on in all future versions of the IDE.

See Also:

`Def Page; Def LPrint;` Inside Macintosh: *Imaging with QuickDraw*

Print**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

Print [@(col,row)|%(h,v)][printItem [{,|;}[printItem]...]]
Print [@(col,row)|%(h,v)][Point [{,|;}[Point]...]]

```

Revision:

May 5, 2000 (Release 3)

Description:

Use this statement to put text out to the current window or to the currently Route'd device. The text is printed using the window's or printer's current font ID, font size, text style, text mode and foreground color (see the `Text`, `Color` and `Long Color` statements). The parameters are interpreted as follows:

When the item to be printed is a point, FB^3 makes special provisions during the print process.

```

Dim mousePos As Point
Call GetMouse(mousePos)
Print mousePos

```

The output looks like this (194x,167y)

- @(col,row)|%(h,v)

This specifies where the first printed character should appear within the window or the printed page. If you use the @(col,row) variant, then *col* and *row* represent the text column and row where the first character should appear; the exact pixel location depends on the current font ID and font size. If you use the %(h,v) variant, then *h* and *v* represent horizontal and vertical pixel positions; the first printed character is positioned with its lower-left corner at point (h,v). If you don't specify either variant, printing begins at the window's or printer's current pen position.

- *printItem*

This can be any of the following:

<i>printItem</i>	<i>Description</i>
a string expression	The string is printed. If the string includes a carriage-return character (ASCII character 13), the character causes the pen to move down to the beginning of the next line.
a numeric expression	The decimal value of the number is printed. A space character is always printed after the number; if the number is non-negative, a space character is also printed before the number. FutureBASIC formats the number in a reasonable way; if you need the number to appear in a special format, use string functions such as <i>Using</i> , <i>Hex\$</i> , <i>Str\$</i> , etc.
a Pointer or Handle variable	The variable's value is interpreted as an address, which is then printed as a numeric expression (see above).
<i>Tab</i> (<i>position</i>)	Sufficient space characters are printed until the current line contains <i>position</i> -1 characters. (If there are already more than <i>position</i> -1 characters on the line, <i>Tab</i> does nothing.) This is usually done to help line up several lines of text into neat columns. Note that this effect looks best if you're using a monospaced font.
<i>Spc</i> (<i>numSpaces</i>)	<i>numSpaces</i> space characters are printed. This has the same effect as printing the string expression <i>Space\$</i> (<i>numSpaces</i>).

- { , ; }

You must use a comma or a semicolon to separate each pair of *printItem*'s; you can also optionally put a comma or semicolon at the end of the *Print* statement (following the last *printItem*).

A comma causes space characters to be printed until the total number of characters on the line is a multiple of the current "tab field width." The default tab field width is 16; you can set it to other values using the *Def Tab* statement. Commas are usually used to help line up several lines of text into neat columns, or just to put some space between consecutive *printItem*'s.

A semicolon does not cause any spaces to be inserted between consecutive *printItem*'s. Use this when you want *printItem*'s to be printed as close together as possible.

Normally, *Print* moves the pen down to the beginning of the next line after the last *printItem* is printed. However, if you put a comma or a semicolon at the end of the *Print* statement, the pen remains to the right of the last *printItem*, and is not moved down to the next line. This allows you to continue printing on the same line using a subsequent *Print* statement.

Note that you can use *Print* without any parameters, like this:

Print

This simply causes the pen to move down to the beginning of the next line; it effectively "prints" a carriage-return character. This is useful for putting blank line(s) between other lines of text.

Line wrap, scrolling and page-eject

By default, if the `printItem`'s reach the right edge of the window or the printer page, the `Print` statement automatically “wraps the line”; that is, it moves the pen down to the beginning of the next line to continue printing the remaining `printItem`'s. However, this behavior can be altered using the `Width` statement; see the `Width` statement for more information.

If you're printing to a window, and the `Print` statement causes the pen to move below the bottom of the window, the window's contents are scrolled up so that the newly printed text will be visible.

If you're printing to the printer, and the `Print` statement causes the pen to move below the bottom of the printer page, the page is ejected and printing continues at the top of the next page.

Console behavior:

When you use the Console runtime, `Print` behaves as follows:

- The `[@(col,row)]` parameters are ignored.
- `Print` switches to the Text Window before printing (unless output has been routed to the printer or `Print%` is used). You can use `Print%` to display text in the Graphics Window at an exact pixel position. Text printing will continue in the graphics window until a carriage return is encountered. You may suppress carriage returns using the semicolon character.
- The `Width` statement has no effect. `Print` always wraps the line at the right edge of the window (or the printer page).

Note:

Text which is displayed using the `Print` statement is not automatically refreshed, unless you're using the Console runtime. To display text which is automatically refreshed, consider using `Edit Fields` (see the `Edit Field` statement).

See Also:

`Text`; `Color`; `Long Color`; `Width`; `Def Tab`; `Using`; `Space$`; `Edit Field`

Print# statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Print# deviceID, [printItem [{,|;}[printItem]...]]
```

Description:

This statement writes information formatted as text to the open file or serial port specified by *deviceID*. The list of *printItem*'s is interpreted and formatted the same way as in the `Print` statement. `Print#` normally writes a carriage-return character (ASCII character 13) after writing the final *printItem*; to inhibit this behavior, put a comma or a semicolon at the end of the `Print#` statement.

`Print#` is typically used to write data which is to be viewed later in a text editor or word processing program; or to write data which is to be read later by the `Input#` statement. It generally formats its output differently than the `Write#` and `Write File` statements, which are better suited for transferring the contents of memory directly to the device. For example, consider this sample program fragment:

```
a% = -1623
Print #1, a%
Write #1, a%
```

In this example, the `Print#` statement formats the number as text, and puts out 7 bytes, as follows:

```
00101101 (ASCII code for "-")
00110001 (ASCII code for "1")
00110110 (ASCII code for "6")
00110010 (ASCII code for "2")
00110011 (ASCII code for "3")
00100000 (ASCII code for a space character)
00001101 (ASCII code for a carriage-return character)
```

On the other hand, the `Write#` statement puts out the binary contents of *a%*. In memory, the short integer `-1623` is stored as `1111100110101001`; therefore, the `Write#` statement puts out these two bytes:

```
11111001 10101001
```

See Also:

`Print`; `Input#`; `Write#`; `Write File`; `Open`; `Def Open`, `Route`

Print Using

statement

See the `Print` statement and the `Using` function.

Proc**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
procAddress& = Proc "label"
```

Description:

This function looks for the indicated *label* in your program, and returns a memory address which can be used to call the instructions which follow *label*. Typically, *label* will indicate the beginning of an `EnterProc` procedure which is to be used as a “callback procedure” by a MacOS Toolbox routine.

- In `cpu68K` compiles, `Proc` returns the memory address of the first program instruction that follows the label. This is the same as the address returned by the `Line` function.
- in `cpuPPC` compiles, `Proc` returns a “universal procedure pointer.” This allows MacOS Toolbox routines in PPC machines to correctly use the referenced `EnterProc` procedure as a callback procedure.

See Also:

```
Call; EnterProc; Line; @Fn
```


PStr\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
string$ = PStr$(address&)
```

Description:

This function returns the string which is located at the indicated *address&* in memory; *address&* must be a long-integer expression or a `Pointer` variable.

The data at *address&* should be a string in “Pascal format,” which is the string format used by FB^3 string variables and by MacOS Toolbox string parameters. In Pascal format, the first byte is interpreted as a number in the range 0 through 255 which indicates the length of the string’s text; this length byte is immediately followed by the text of the string.

See Also:

```
PStr$ statement; Str#
```

PStr\$ statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
PStr$(addressVar&) = string$
```

Description:

This statement changes the value of *addressVar&*, setting it to the address of *string\$*. *addressVar&* must be a long-integer variable or a `Pointer` variable; *string\$* must be a string variable or a literal string in quotes.

If *string\$* is a string variable, the `PStr$` statement is equivalent to this:

```
addressVar& = @string$
```

If *string\$* is a literal string in quotes, then *addressVar&* is set to an address in FutureBASIC's heap where the literal string is stored. The `PStr$` statement is the only way to obtain the address of a literal quoted string.

Note:

The `PStr$` keyword can also be used with the `Read` statement, to obtain the address of a string specified in a `Data` statement.

See Also:

`PStr$` function; `Read`

Push

statements

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

Push (var)
Push {Byte|Word|Long} (address&)

```

Description:

These statements put 1, 2 or 4 bytes onto the CPU stack, and adjust the stack pointer by a corresponding amount. `Push` is the opposite of `Pop`.

If you use the first syntax, the data is copied from *var*, which must be a variable. The number of bytes put onto the stack depends on the data type of *var*:

<i>var type</i>	<i>Bytes put onto stack</i>
byte (signed or unsigned)	1
short integer/ Word (signed or unsigned)	2
long integer (signed or unsigned); Pointer ; Handle	4

(No other variable types are valid with `Push (var)`.)

If you use the second syntax, the data is copied from the memory which begins at *address&*, which must be a long integer expression or a **Pointer** variable. The number of bytes put onto the stack depends on which keyword you use:

<i>Keyword</i>	<i>Bytes put onto stack</i>
Byte	1
Word	2
Long	4

Note:

In CPU68k compiles, `Push` always adjusts the stack pointer by an even number of bytes. If you use `Push (byteVar)` or `Push Byte (address&)` in a CPU68k compile, the stack pointer will be adjusted by 2 bytes, even though only one byte is copied from *byteVar* or *address&*.

`Push` is meant for careful use by advanced programmers. Your system can crash if the stack pointer is not adjusted carefully.

See Also:

`Pop`

Put Preferences

statement✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Put Preferences prefFilePath$, prefRecord
```

Revision:

February, 2002 (Release 6)

Description:

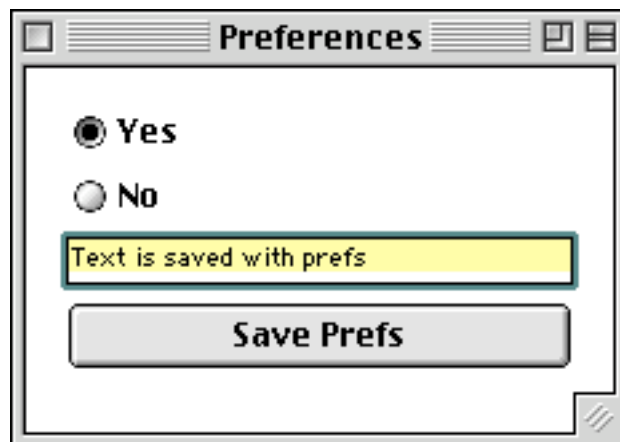
This statement locates the named preferences file and writes the contents of the file from the indicated preferences record. If the file does not exist, it is created. This routine places the file in the preferences folder and works regardless of what system software is in use. The file may contain a folder in the form foldername:fileanme where a colon separates the folder name(s) from the file name.

In order to use any of the `Preferences` commands you must include the proper header file using the following statement:

```
Include "Subs PrefsFile.Incl"
```

Example:

The following example shows how to manage a set of preferences using the new `Preferences` commands. A Boolean value and a text string are saved in the preference file when the "Save Prefs" button is clicked. If you run the program a second time, you will see that modified prefs were properly stored and retrieved.



```

Include "Subs PrefsFile.Incl"

Begin Globals

Begin Record prefsRecord
    Dim prefChoice As Boolean
    Dim prefText   As Str255
End Record

Dim gPref As prefsRecord

End Globals

Local Fn readPreferences
    // set up defaults in case there is nothing to read (as
    // would be the case with the first run of the program)

    gPref.prefChoice = _zTrue
    gPref.prefText   = "Text is saved with prefs"

    // read the prefs on top of the defaults
    Get Preferences "My Pref File",gPref

End Fn

Local Fn savePreferences
    gPref.prefChoice = (Button (1) = _markedBtn)
    gPref.prefText   = Edit$(1)
    Put Preferences "My Pref File",gPref
End Fn

Local
Dim r As Rect
Local Fn setUp
    SetRect(r,0,0,220,136)
    Window 1, "Preferences",@r
    SetRect(r,16,16,Window(_width) - 16,32)
    Button 1,1, "Yes",@r,_radio
    OffsetRect(r,0,24)
    Button 2,1, "No",@r,_radio

    OffsetRect(r,0,24)
    Edit Field 1,"",@r
    OffsetRect(r,0,24)
    r.botTom = r.botTom + 8
    Button 3,1,"Save Prefs",@r

    If gPref.prefChoice Then Button 1,2 Else Button 2,2
    Edit$(1) = gPref.prefText
End Fn

```

FUTUREBASIC REFERENCE

```
Local
Dim action,reference
Local Fn dodialog
    action    = Dialog(0)
    reference = Dialog(action)
    Long If action = _btnclick

        Select reference
            Case 1      //yes
                Button 1,2
                Button 2,1
            Case 2      //no
                Button 1,1
                Button 2,2
            Case 3      //save prefs
                Fn savePreferences
                gFBquit = _zTrue
        End Select
    End If
End Fn

On Dialog Fn doDialog
Fn readPreferences
Fn setUp

Do
    HandleEvents
Until gFBquit
```

See Also:

Get Preferences; Kill Preferences; Menu Preferences

Random

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Random[ize] [expr]
```

Description:

This statement “seeds” the random number generator: this affects the sequence of values which are subsequently returned by the `Rnd` function and the `Maybe` function.

The numbers returned by `Rnd` and `Maybe` are not truly random, but follow a “pseudo-random” sequence which is uniquely determined by the seed number. If you use the same seed number on two different occasions, you’ll get the same sequence of “random” numbers both times. For example:

```
Cls
For i = 1 To 2
  Randomize 325
  For j = 1 To 10
    Print Rnd(50),
  Next
  Print
Next
```

The program above seeds the random number generator twice with the same number (325). If you run this program, you’ll find that it produces the same sequence of 10 random numbers after each seeding.

Seeding the random number generator with a pre-specified number can be useful in cases where you specifically want to produce a repeatable sequence of random numbers. In most cases, however, you will probably prefer a sequence that is unpredictable. In that case, you should omit the `expr` parameter. When you execute `Random` without any `expr` parameter, the system’s current time is used to seed the random number generator. Since the system clock changes very quickly, it’s essentially impossible to predict what value will be used as the seed—this is the best way to get the “most random” random numbers.

Normally, you will execute `Random` only once in your program, unless you wish to repeat a specific sequence of random numbers. If you don’t execute `Random` at all, the random number generator is seeded with the current tick count. Remember that re-seeding with the same number will cause your program to generate the same sequence of random numbers each time it’s run.

See Also:

`Rnd`; `Maybe`

Randomize

statement

This is a synonym for the `Random` statement

Ratio

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

Ratio *h*, *v*

Description:

This statement affects the width and height of shapes subsequently drawn with the `Circle` statement. By suitably setting the *h* and *v* parameters, you can draw ellipses with any aspect ratio.

Each of *h* and *v* can range in value from -128 to +127. The *h* parameter affects the width of the ellipse, and the *v* parameter affects its height. When you subsequently execute a `Circle` statement with a `radius` parameter of `radius`, a circle or ellipse is drawn which has these dimensions:

Shape's width = $2 * \text{radius} * (1 + h / 128)$

Shape's height = $2 * \text{radius} * (1 + v / 128)$

We can notice a few consequences of these formulas:

- If *h* and *v* are equal, a circle is drawn. If *h* and *v* are different, an ellipse is drawn.
- If $h > v$, the ellipse is wider than it is tall. If $v > h$, the ellipse is taller than it is wide.
- Negative values of *h* or *v* cause the corresponding dimension to shrink. Positive values cause the corresponding dimension to grow (up to about twice its original size).
- Specifying `Ratio 0,0` “resets” the ratios: subsequent `Circle` statements will draw a circle with the indicated radius.

After you execute a `Ratio` statement, the indicated *h* and *v* values remain in effect for all subsequent `Circle` statements (in all windows), until another `Ratio` statement is executed.

See Also:

`Circle`

Read

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Read {var|PStr$(addressVar&)}[, {var|PStr$(addressVar&)}...]
```

Description:

This statement reads one or more of the items listed in one or more `Data` statements. If you specify `var` (which must be either a numeric variable or a string variable), the data item's value is stored into `var`. If you specify `PStr$(addressVar&)`, the item is interpreted as a string and its address is stored into `addressVar&` (which must be a long-integer variable or a `Pointer` variable).

Each `var` or `PStr$` that you specify causes one data item to be read. The first time your program executes a `Read` statement, the first item in your program's first `Data` statement is read. Every time a `var` or `PStr$` is encountered in any `Read` statement, the next data item is read, until all items in all your program's `Data` statements have been exhausted. The number of `var` or `PStr$` specifications in a `Read` statement does not need to match the number of items in a `Data` statement; however, the total number of read requests should not exceed the total number of items in all `Data` statements (unless you use the `Restore` statement, which allows you to re-use data from previous `Data` statements).

Example:

```
Data 1,2
Data 3,4
Data 5,6
For i = 1 To 2
  Read a, b, c
  Print a, b, c
Next
```

Program output:

```
1      2      3
4      5      6
```

See Also:

```
Data; Restore
```

Read#**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Read# deviceID, {recVar|numVar|strVar$; len} ↯
        [, {recVar|numVar|strVar$; len}...]
```

Description:

This statement reads data from the open file or serial port specified by *deviceID*, and stores the data into the indicated variable(s).

You can read the data into record variables (*recVar*), into numeric variables (*numVar*) or into string variables (*strVar\$*), or any combination of these. If you specify *recVar* or *numVar*, the statement reads a number of bytes equal to the size of the variable, and stores the bytes directly into the variable's location in memory, without doing any data conversion. If you specify *strVar\$*; *len*, the statement reads *len* bytes, and stores them into *strVar\$* as a Pascal string of length *len* (*len* can be any numeric expression, but its value should not exceed 255). The read operation begins at the current location of the “file mark,” and the file mark is advanced as each item is read. If Read# attempts to read past the end of the file, FB^3 generates an error.

Because Read# copies the file's bytes directly into memory without conversion, it's best suited for reading “binary” information, such as that created by the Write# statement. To read file data which is formatted as text, it's usually better to use the Input# statement. To read an arbitrary number of bytes into a block of memory, use the Read File statement.

See Also:

Write#; Input#; Read File; Eof; Open; SizeOf

Read Dynamic

statement✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**`Read Dynamic deviceID, arrayName`**Revision:**

May, 2001 (Release 5)

Description:

Use `Read Dynamic` to read the contents of a dynamic array from a disk file. Before executing this read statement, you must dimension the dynamic array.

Example:

The following example creates and fills a dynamic array, writes the array to disk, then reads it back into memory.

```

Dim x
Dynamic myAry(_maxLong)

For x = 1 To 100
    myAry(x) = x
Next

Open "O",#1,"Dynamic Array Test"
Write Dynamic #1,myAry
Close #1

Kill Dynamic myAry

Open "I",#1,"Dynamic Array Test"
Read Dynamic #1,myAry
Close #1

For x = 1 To 10
    Print myary(x)
Next

Kill "Dynamic Array Test"

```

See Also:`Compress Dynamic, Dynamic, Write Dynamic`

Read Field

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Read Field [#]deviceID, HandleVar
```

Description:

This statement creates a new relocatable block in memory, reads data from the open file or serial port specified by *deviceID* into the new block, and returns a handle to the block into *HandleVar*, which must be a long-integer variable or a *Handle* variable.

The data in the file (or coming in through the serial port) must be in a particular format in order to be read properly. The first 4 bytes (at the current “file mark” position) must be a long integer which indicates the size of the block to create. This should be immediately followed by the data which is to go into the block. This is the format in which the *Write Field* statement writes to a file; almost always, the data you read with *Read Field* will have been created using a *Write Field* statement.

Note:

Your program is responsible for disposing of the handle returned in *HandleVar* when you’re finished using the block. You can use a statement such as *Def DisposeH* or *Kill Field* to do this.

See Also:

```
Write Field; Read File; Open; Kill Field; Def DisposeH
```

Read File

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Read File [#]deviceID, address&, numBytes&
```

Description:

This statement reads *numBytes*& bytes from the open file or serial port specified by *deviceID* (starting at the current “file mark” position), and copies them into memory starting at the address specified by *address*&. This is the fastest way to read large amounts of data from a file; it’s also well suited for reading data whose format you may not know in advance.

Example:

This program fragment quickly loads an array with the data read from a file. It’s assumed that the binary image of the array was previously saved to the file using a statement like `Write File` (see the example accompanying the `Write File` statement).

```
_maxSubscript = 200
Dim myArray%(_maxSubscript)
arrayBytes& = (_maxSubscript+1) * SizeOf(Int)
Read File #1, @myArray%(0), arrayBytes&
```

Note:

If `Read File` attempts to read past the end of the file (because *numBytes*& was too large), FB^3 generates an error.

See Also:

`Open`; `Read#`; `Read Field`; `Write File`

Rec**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
currentRecord = Rec(fileID)
```

Description:

This function returns the record number of the record where the “file mark” is currently located, in the open file specified by *fileID*. The first record in the file is considered Record #0.

To calculate the record number, `Rec` uses the record length that was specified in the `Open` statement when the file was opened. If no record length was specified, a default length of 256 bytes is used. If you specified a record length of “1” when you opened the file, `Rec` returns the file mark’s byte position within the file.

Note:

To determine the file mark’s offset from the beginning of the record, use the `Loc` function.

See Also:

`Record`; `Loc`; `Open`

Record statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Record [#] fileID, recordNum [, positionInRecord]
```

Description:

This statement sets the “file mark” position, in the open file specified by *fileID*. The position of the file mark determines the location in the file where the next input or output operation will occur.

If you omit the *positionInRecord* parameter, **Record** places the file mark at the beginning of the record indicated by *recordNum* (the first record in the file is considered Record #0). If you specify *positionInRecord*, the file mark is placed at an offset of *positionInRecord* bytes past the beginning of the indicated record.

Record uses the record length that was specified in the **Open** statement when the file was opened. If no record length was specified, a default length of 256 bytes is used. If you specified a record length of “1” when you opened the file, you can set the file mark to a particular byte offset from the beginning of the file using a statement like this:

```
Record [#] fileID, byteOffset&
```

Note:

The file mark position is also moved automatically after every input or output operation on the file.

See Also:

Rec; **Loc**; **Open**; **Lof**

Register (A5)**function (68K only)**

*✗ Appearance**✓ Standard**✓ Console*

Syntax:

```
currentA5 & = Register (A5)
```

Description:

This function returns the value of the system global variable *currentA5*.

The *currentA5* value is a pointer to a location within the memory space allotted for your program; every currently running program has its own individual *currentA5* value. *currentA5* points to the first byte of your program's application parameters. The memory space immediately preceding the location pointed to by *currentA5* contains your program's global variables. See the "Introduction to Memory Management" chapter in Inside Macintosh: Memory for more information.

See Also:

Inside Macintosh: *Memory*

Register On/Off

statements

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

Register On
Register Off
Bool = _registerVars

```

Revision:

June, 2001 (Release 5)

Description:

The constant `_registerVars` is set up by the editor before compilation begins. The constant is zero (`_false`) if register variables are currently turned off and -1 (`_zTrue`) if they are on. You may not change the status of the constant by overriding. It is for information purposes only.

These are non-executable statements which affect the way the compiler treats variables which are defined farther down in the program.

The `Register On` statement instructs the compiler to attempt to use CPU registers to store the values of subsequently-encountered variables. The `Register Off` statement instructs the compiler to use addressable memory (RAM) to store the values of all subsequently-encountered variables. These statements are global in scope; they affect all variables that are defined below the statement, whether they're defined in a local function or in the "main" part of your program. Your program can include several `Register On` and `Register Off` statements, if you wish to exercise finer control over how specific variables are used. If you don't specify either statement, the default condition is determined by how you've set this option in the "Compiler Preferences" dialog.

The advantage of using register-based variables is speed. Many operations are faster by orders of magnitude when they use register-based variables. Only 1-byte, 2-byte and 4-byte variables can be stored in CPU registers. There are only a limited number of registers available, so it's possible that not all of your variables will be register-based even if you specify `Register On`. When using `Register On`, you should define your most frequently used variables first (just below the `Register On`); this will increase their chances of being assigned to a CPU register.

The advantage of using RAM-based variables is versatility. There are certain operations that can't be performed using register-based variables; in particular, any operation that makes reference to a variable's address must use a RAM-based variable. This includes using expressions like `VarPtr(myVar)` and `@myVar`, and passing the variable to certain statements and functions which need to return a value into the variable. The compiler will generate an error if you attempt to use a register-based variable in such situations.

Floating Point Registers

While the specific set of registers used for most operations is designed to handle whole numbers, there is an additional set of registers on the PPC processor set aside for floating point values. It is important to note that all floating point registers are double precision. There is no advantage to attempting to use a single precision variable. In fact, the conversion to double precision may make it slower. This same feature is true of all math operations. Single precision variables are converted to double precision, the operation is performed, then the variable is converted back to its single precision state.

In order to force a float into a register, you must dimension it between the `Local` and the `Local Fn` statements. The following example shows register and non-register floats in a local function.

```

Local
Dim myRegisterVar#   'in register
Dim myVar As Double 'in register
Local Fn anything
    Dim myRamFloat#   'not in register
    Dim notInRegister As Double   'not in register
    Rem Do anything
End Fn

```

Note:

You can also use the `Dim` statement to selectively force certain variables to be RAM-based, even when `Register On` is in effect. Here are some items that force variables to be located in RAM:

```

Dim @var
Dim var&;0,hi%,lo%
Dim var.8

```

See Also:

`Dim`

Rem**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
[statement] Rem [remarks]
[statement] '[remarks]
[statement] //[remarks]
[statement] /*[blockRemarks]*/ [statement]
```

Description:

The `Rem` statement (and its variations) provide a way for you to insert remarks into the source code which are ignored by the compiler. Remarks have absolutely no effect on how your program runs, but they can be very useful in helping readers to understand how your code works.

If you use the `Rem` keyword, or the apostrophe (`'`) or double-slash (`//`) token, everything following it on the same line is treated as a remark; therefore, it's not possible to put a non-remark statement after *remarks* on the same line.

The apostrophe and double-slash variations work identically to each other. In the FB^3 Editor, remarks that begin with the apostrophe, the double-slash, or the `/*` token are automatically tabbed to the Remarks column of the Editor window if they're preceded by a *statement*. Remarks that begin with `Rem` are not tabbed.

When you use the `/*` and `*/` tokens, the `/*` indicates the beginning of the remarks, and the `*/` indicates the end of remarks. Because it uses two tokens, this variation allows you to do some things you can't do with other variations. Specifically:

You can write remarks that span multiple lines, using only one pair of tokens. For example:

```
/* This is the first line of remarks.
   The remarks continue on this line.
   This is the last line of remarks. */
```

One disadvantage of using the `/*` and `*/` tokens is that you have to make sure you don't inadvertently include a `*/` token in the middle of your remarks. If you do, the compiler will assume that the remarks end there. For example:

```
/* This is the first line of remarks.
   We have /*accidentally*/ put a remark-closing Token
   in the middle of the remarks.
  */
```

When the compiler encounters this, it interprets the `*/` token following `accidentally` as the end of the remarks, and it attempts to compile the remainder of that line and the following line, leading to compile errors.

Note:

You cannot put remarks after a `Data` statement on the same line. Everything following the `Data` keyword on the same line is considered part of the `Data` statement.

See Also:

`Data`

Rename

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Rename oldPath$ {To|,} newFileNameOnly$ [,refNum% [,dirID&]]
```

Description:

Changes the name of a file or folder from the name specified in *oldPath\$* to the name specified in *newPath\$*. The *oldPath\$* and *newPath\$* parameters are used along with *refNum%* and *dirID&* to determine the location of the item to be renamed: see Appendix A: *Specifying Files and Directories*, for more information. You cannot use `Rename` to move an item into a different directory: *oldPath\$* and *newPath\$* must both refer to items in the same directory.

See Also:

Folder; Appendix A: *Specifying Files and Directories*

Reset

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

`Reset`

Description:

This statement closes all files and devices that have been opened with the `Open` statement. `Reset` is functionally identical to using the `Close` statement without any parameters.

See Also:

`Open`; `Close`

Resources

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Resources "[pathname]" [,"ttttcccc"]
```

Revision:

May, 30, 2000 (Release 3)

Description:

This is a non-executable statement that performs two main functions:

- It specifies your program's file type and creator;
- It identifies an existing file (*pathname*) containing resources that should be copied into the resource fork of your application file or code resource file, when FB^3 builds your file.

The `Resources` statement is optional. If you don't include it in your program, FB^3 builds an application file of type "APPL", with creator signature "xxxx", containing a standard set of resources.

Resources may also be added by dragging a resource file into the project manager window.

You can include multiple `Resources` statements in your program. FB^3 uses the first encountered `Resources` statement to determine the file type and creator.

The *pathname* parameter can be a full or partial pathname which specifies a file that contains resources; or which specifies an alias to such a file. If you use a partial pathname (for example, a simple file name), the path is assumed to be relative to your project folder. When FB^3 builds your application file, it copies all the resources from the *pathname* file into the resource fork of the file that it builds. Note that if you don't specify the *pathname* parameter, you still must specify an (empty) pair of double-quotes.

Using multiple Resources statements

It's sometimes convenient to have FB³ copy the resources from several different resource files. You can accomplish this by including multiple `Resources` statements in your program, each specifying a different *pathname* parameter. If your program includes multiple `Resources` statements, the second and all subsequent `Resources` statements should not specify any parameter other than *pathname*.

If your program includes multiple `Resources` statements, there is the possibility of a "collision" between resources. This happens when a resource in one *pathname* file has the same type and same ID number as a resource in another *pathname* file. When this happens, the resource that was encountered later replaces the resource that was encountered earlier, when FB³ builds your file. You should keep this in mind when deciding in what order to place your `Resources` statements.

New feature in Release 3: If your resource has an ID of 32512-32767, the compiler will renumber it when it sees the conflict. That way, you can refer to it by name and pick it up with `GetNamedResource`. This is important for those that want to distribute source code with required resources.

Example: You have two resource files in your project.

```
myRes1.rsrc
myRes2.rsrc
```

myRes1.rsrc contains:

```
_ "PICT" ID 501      Name "One"
_ "PICT" ID 502      Name "Two"
_ "PICT" ID 32512    Name "Fred"
```

myRes2.rsrc contains:

```
_ "PICT" ID 501      Name "OneOne"
_ "PICT" ID 502      Name "TwoTwo"
_ "PICT" ID 32512    Name "Barney"
```

Your final application will contain:

```
_ "PICT" ID 501      Name "OneOne"  <- Note that OneOne replaced One
_ "PICT" ID 502      Name "TwoTwo"  <- Note that TwoTwo replaced Two
_ "PICT" ID 32512    Name "Fred"    <- First version of 32512 saved
_ "PICT" ID 32513    Name "Barney"  <- Second version of 32512 renumbered
```

Some useful resources

When building an application, FB^3 automatically includes a standard set of resources that applications require. You can enhance your application by also including resources like the following (all of which can be created using ResEdit):

- **vers** “vers” resources with ID’s 1 and 2 contain version information which is visible in Finder windows and in the “Get Info” window. See the “Finder Interface” chapter of Inside Macintosh: *Macintosh Toolbox Essentials*, for more information.
- **SIZE** “SIZE” resources with ID’s -1, 0 and 1 contain important information about your application’s memory size, what kinds of events it can respond to, and more. FB^3 always includes a “SIZE” resource whenever it builds an application, but you may want to override the features in the default “SIZE” resource by providing one of your own. See the “Event Manager” chapter of Inside Macintosh: *Macintosh Toolbox Essentials*, for more information.
- **BNDL, FREF, ICN#** Use these resources to assign special icons to your application and to the documents that your application creates. These resources also determine what kinds of documents can be dragged to you application’s icon in the Finder. See the “Finder Interface” chapter of Inside Macintosh: *Macintosh Toolbox Essentials*, for more information.

See Also:

Call <resource>; Button (custom CDEF’s); EnterProc%

Restore statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:****Restore** [*expr*]**Description:**

This statement is used in conjunction with the `Data` and `Read` statements. It resets an internal pointer which tells FB^3 where to find the next `Data` item to read. This allows your program to `Read` the same `Data` item(s) more than once if necessary.

If you omit the *expr* parameter, the `Data` pointer is reset to point to the first item in the first `Data` statement. If you specify *expr*, the `Data` pointer is reset to point to the item immediately following the *expr*-th item. Thus `Restore 1` points to the second item; `Restore 2` points to the third item; and so on.

Example:

```

Data Able, Baker
Data Kane, Dread
Data Echo
For i = 1 To 5
    Read x$
    Print x$,
Next
Print
Restore 3
Read x$, y$
Print x$, y$

```

Program output:

```

Able Baker Kane Drea Echo
Dread Echo

```

See Also:

`Read`; `Data`

Return

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Return ["label"]
```

Revision:

October 2, 2000 (Release 4)

Description:

You should include at least one `Return` statement in every subroutine that is called by a `Gosub` statement. `Return` causes the subroutine to “exit”; that is, it causes execution to continue at the statement following the `Gosub` that called the subroutine.

You may also return to a specific location using `Return "label"`. This pops the return address from the stack, then jumps to the requested address.

See Also:

`Gosub`

Right\$ and Right\$\$

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
subString$      = Right$(string$, numChars)
subContainer$$ = Right$(container$$, numChars)
```

Revision:

May 30, 2000 (Release 3)

Description:

This function returns a string or container consisting of the rightmost *numChars* characters of *string\$* or *container\$\$*. If *numChars* is greater than the length of *string\$* or *container\$\$*, the entire *string\$* or *container\$\$* is returned. If *numChars* is less than 1, an empty (zero-length) string is returned.

Note:

You may not use complex expressions that include containers on the right side of the equal sign. Instead of using:

```
c$$ = c$$ + Right$(a$,10)
```

Use:

```
c$$ += Right$(a$,10)
```

Example:

```
Print Right$("Nebraska", 3)
```

Program output:

ska

See Also:

Left\$; Mid\$; InStr

Rnd**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
randomInteger% = Rnd(expr%)
```

Description:

This function returns a pseudo-random long integer uniformly distributed in the range 1 through *expr&*. The *expr&* parameter should be greater than 1, and must not exceed 65536. If the value returned is to be assigned to a 16-bit integer (*randomInteger%*), *expr&* should not exceed 32767. The actual sequence of numbers returned by **Rnd** depends on the random number generator's "seed" value; see the **Random** statement for more information. Note that **Rnd**(1) always returns the value 1.

Example:

To get a random integer between two arbitrary limits *min* and *max*, use this statement:

```
randomInteger = Rnd(max - min + 1) + min - 1
```

(Note: *max* - *min* must be less than or equal to 65536.)

To get a random fraction, greater than or equal to zero and less than 1, use this statement:

```
frac! = (Rnd(65536)-1)/65536.0
```

To get a random long integer in the range 1 through 2,147,483,647, use this statement:

```
randomInteger& = ((Rnd(65536) - 1)<<15) + Rnd(32767)
```

See Also:

Random; **Maybe**

Route statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

To re-route text and graphics:

Route [#] `_toPrinter`

Route [#] `_toScreen`

Route [#] `_toBuffer` [+ 0... 4]

Route [#] `_toAppleScript`

To re-route text:

Route [#] `serialPort`

Route [#] `fileID`

Revision:

May, 2001 (Release 5)

Description:

This statement causes text printed by subsequent `Print` statements to be sent to the indicated device. If you specify `_toScreen` or `_toPrinter`, subsequent drawing commands are also sent to the indicated device. If you specify `_toBuffer`, only text printing commands are sent to the indicated device.

Using Route `_toPrinter`

`Route _toPrinter` opens a new print job unless a print job is already open. When you first open a new print job, the printer adopts the font and graphics characteristics that are in effect in the current screen window. Subsequent `Print` statements, as well as graphics commands such as `Box`, `Circle` and `Plot`, are sent to the printer. Subsequent statements which affect the appearance of text and graphics, such as `Text`, `Pen` and `Color`, apply to the printed output. Subsequent QuickDraw Toolbox commands such as `FrameRect` apply to the printed output.

Note: To actually put out the current printer page, use the `Clear LPrint` statement after sending text and drawing commands to the printer.

Using Route `_toScreen`

After using `Route` to direct output to the printer, or to a serial device or a file, you can use `Route _toScreen` to re-direct output back to the screen window again.

Note: Statements which affect the appearance of text and graphics, such as `Text`, `Pen` and `Color`, apply separately to the screen window and to the printer. The settings in one device don't affect the settings in the other, except when you first open a new print job.

Using Route serialPort and Route fileID

These statements cause text printed by subsequent `Print` statements to be sent to the specified open device. They don't affect the destination of graphics commands. This group of statements:

```
Route deviceID
Print itemList1
Print itemList2
:
Print itemListN
Route _toScreen
```

has the same effect as this group of statements:

```
Print #deviceID, itemList1
Print #deviceID, itemList2
:
Print #deviceID, itemListN
```

Using Route _toBuffer

You can use the FB `Route _toBuffer` statement to print information directly to a handle. You may use any one of five handles by setting the route statement in the range of `_toBuffer` through `_toBuffer + 4`. When routed to a buffer, only text commands are sent to the handle. Graphic commands are ignored.

Information printed to a buffer ends up in one of five handles stored as globals in the FB runtime. This array is named `gFBbuffer(n)` where `n` is a numeric expression in the range of 0...4. The following example prints text to buffer number 2.

```
Route _toBuffer + 2
Print "Hello"
Print "Goodbye"
Route _toScreen
```

When this snippet has been executed, `gFBbuffer(2)` will contain a handle that points to a moveable block of 14 bytes:

```
Hello<CR>Goodbye<CR>
```

Note that carriage returns are added or suppressed in writing to a buffer just as they would be in writing to a file or to the screen.

If you want to dispose of a buffer, use `Def DisposeH`.

```
Def DisposeH(gFBbuffer(2))
```

Using Route _toAppleScript

Use this routing to build a buffer of AppleScript commands. All subsequent `Print` statements are sent to this buffer which may later be invoked using the `Usr AppleScript`'s set of commands. Errors in a script will be reported when the script is executed. FutureBASIC has no means for determining the correctness of your script. Errors in AppleScript syntax will be reported when the script is executed.

Console behavior:

When you use the Console runtime, built-in graphics statements such as `Box`, `Circle` and `Plot` are always directed to the Graphics Window; using `Route _toPrinter` won't cause such commands to be directed to the printer. If you want to send graphics to the printer while using the Console, first execute `Route _toPrinter`, then use QuickDraw Toolbox routines such as `FrameRect`, `FrameOval`, `LineTo`, etc.

Note:

The `Window` statement and the `Window Output` statement implicitly execute a `Route _toScreen` statement. Also, any user action which activates a screen window (such as clicking on its title bar) will implicitly cause a `Route _toScreen`. If you want to make sure that output is not inadvertently directed to the screen window, you should check for `_wndActivate` events in your dialog-event handling routine, and re-direct the output as necessary.

See Also:

`Open`; `Print`; `Print#`; `Clear LPrint`; `Close LPrint`; `Text`; `Pen`; `Color`; `Long Color`; `Box`; `Circle`; `Plot`; `On Dialog`; *Inside Macintosh: Imaging with QuickDraw*

Run	statement		
✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>	

Syntax:

```
Run path$ [,refNum% [,dirID&]]
```

Description:

This statement launches the application specified by *path\$*, *refNum%* and *dirID&*, and puts it into the foreground, making it the active process. The program that launched the application does not quit, but is put into the background.

See Appendix A: *Specifying Files and Directories*, to see how the *path\$*, *refNum%* and *dirID&* parameters are interpreted.

See Also:

Inside Macintosh: *Processes*

Runtime

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

Runtime routineName [(arg1 [,arg2 ...])]
                    [statementBlock]
End Fn [= expr]

End Runtime

```

Description:

The FB^3 compiler has been created in such a way that it expects and calls upon specific functions located in the header folder. As an example, the statement `Window Close wNum` is routed to a runtime function named `Runtime FBWindowClose(wNum&)`.

A `Runtime` function behaves much like a local function, but the "Runtime" name indicates to FB that the function is part of the language. You may not add new syntax to FB^3 by creating new `Runtime` functions. Only the compiler can make these calls. But you can modify these pieces of code as you would any source code.

When all of the necessary headers have been compiled and FB^3 is ready to begin examining your source code, the `End Runtime` statement tells FB^3 that it no longer needs to track and record `Runtime` functions.

As you examine the `Runtime` functions, you may notice that only long integer parameters are used. This serves three purposes:

- Long integers can be passed faster than any other type of parameter
- `Runtime` functions know in advance what type of parameter is expected. For instance, a `Runtime` function might expect a string pointer, but never a string.
- The use of long integers insures that variables will fall on long integer boundaries which greatly speeds the execution of PPC code.

Scroll

statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Scroll (h1,v1)-(h2,v2), hPixels%, vPixels%
```

Description:

This statement scrolls the pixels in the rectangle $(h1, v1) - (h2, v2)$, by a horizontal distance of *hPixels%* and a vertical distance of *vPixels%*, in the current output window. Positive values of *hPixels%* scroll to the right; positive values of *vPixels%* scroll downward; use negative values to scroll in the opposite direction(s). `Scroll` is often used in conjunction with scroll bars to view a graphic that is too large to fit entirely in the window.

`Scroll` generates a `_wndRefresh` event, which your program can detect in its dialog-event handling routine the next time a `HandleEvents` statement is executed. Your dialog-event handling routine should respond to this event by redrawing the portion of the window that was left blank by the scroll. If you don't redraw this area explicitly, it will be filled with the window's current background color and pattern.

Console behavior:

When you use the Console runtime, `Scroll` operates only on the Graphics Window. Because the Console runtime refreshes the window automatically, you won't receive a `_wndRefresh` event.

See Also:

`Scroll Button; On Dialog`

Scroll Button

statement

✓ *Appearance*

✓ *Standard*

✗ *Console*

Syntax:

```
Scroll Button [#]idExpr ~
    [, [current] [, [min] [, [max] [, [page] [, [rect] [, type]]]]]
```

Description:

The `Scroll Button` statement puts a new scrollbar in the current output window, or alters an existing scrollbar's characteristics. The parameters are interpreted as follows:

Parameter	Description
<i>idExpr</i>	An integer which identifies the scrollbar. If <code>Abs(idExpr)</code> is different from the ID numbers of all buttons and all other scrollbars in the current window, a new scrollbar is created, and is assigned an ID number equal to <code>Abs(idExpr)</code> . If <code>Abs(idExpr)</code> equals the ID of an existing scrollbar, the scrollbar's characteristics are altered. Use a negative <i>idExpr</i> value to link the scrollbar's action to an edit field (explained below).
<i>current</i>	This sets the current "value" of the scrollbar, which, along with <i>min</i> and <i>max</i> , determines the position of the scrollbar's "thumb." It must be greater than or equal to <i>min</i> , and less than or equal to <i>max</i> .
<i>min</i>	The minimum value that the scrollbar can have. For vertical scrollbars, this corresponds to a thumb position at the top of the bar; for horizontal scrollbars, it corresponds to a thumb position at the left side of the bar. <i>min</i> must be in the range -32768 through +32767.
<i>max</i>	The maximum value that the scrollbar can have. For vertical scrollbars, this corresponds to a thumb position at the bottom of the bar; for horizontal scrollbars, it corresponds to a thumb position at the right side of the bar. <i>max</i> must be in the range -32768 through +32767, and must be greater than <i>min</i> .
<i>page</i>	This specifies the amount by which the scrollbar's value should change when the user clicks in the areas between the thumb and the scrollbar's end-arrows. Must be non-negative.
<i>rect</i>	<p>For scrollbars of type <code>_scrollOther</code>, the <i>rect</i> parameter specifies the rectangle that defines the size and position of the scrollbar. You can specify it in either of two ways:</p> <div style="display: flex; align-items: flex-start;"> <div style="margin-right: 20px;"> <p><code>(x1,y1)-(x2,y2)</code></p> <p><code>rectAddr&</code></p> </div> <div> <p>Coordinates of two diagonally opposite points.</p> <p>Address of an 8-byte rectangle structure. If the specified rectangle is wider than it is tall, the scrollbar becomes a horizontal scrollbar. If the rectangle is taller than it is wide, the scrollbar becomes a vertical scrollbar. The standard recommended width for a vertical scrollbar (or height for a horizontal scrollbar) is 16 pixels.</p> </div> </div> <p>Note: For scrollbars of type <code>_scrollHorz</code> or <code>_scrollVert</code>, the <i>rect</i> parameter is interpreted differently. See below for more details.</p>

type	<p>Specify one of the following:</p> <p><code>_scrollOther</code>: The scrollbar occupies the rectangle specified in the <code>rect</code> parameter.</p> <p><code>_scrollVert</code>: The scrollbar occupies the right edge of the window, and is resized as the window is resized. If you specify a <code>rect</code> parameter when creating the scrollbar, the top of the scrollbar is offset from the top of the window by <code>y1</code> pixels.</p> <p><code>_scrollHorz</code>: The scrollbar occupies the bottom edge of the window, and is resized as the window is resized. If you specify a <code>rect</code> parameter when creating the scrollbar, the left side of the scrollbar is offset from the left side of the window by <code>x1</code> pixels.</p> <p>Note: <code>_scrollVert</code> and <code>_scrollHorz</code> scrollbars can only be put into windows of type <code>_doc</code>, <code>_docZoom</code> and <code>_docNoGrow</code>. If you try to create a <code>_scrollVert</code> or <code>_scrollHorz</code> scrollbar in any other type of window, the scrollbar won't appear.</p>
------	---

To Create a New Scrollbar:

- Choose an `idExpr` value such that `Abs(idExpr)` is different from the ID's of all existing buttons and scrollbars in the window. If you use a negative `idExpr`, and the window contains a multistyled non-static edit field whose ID number is `Abs(idExpr)`, then the actions of the scrollbar are linked to the edit field (explained below).
- Choose initial values for `current`, `min`, `max` and `page`. All of these parameters are optional; any of them that you omit will have the following default initial values:
 - `current` = 0
 - `min` = 0
 - `max` = 255
 - `page` = 16
- If creating a `_scrollOther` scrollbar, specify the `rect` parameter. This parameter is optional if you're creating a `_scrollVert` or `_scrollHorz` scrollbar.
- Specify the type. This parameter is optional; its default value is `_scrollOther`.

To Alter an Existing Scrollbar:

- Set `idExpr` to the ID number of an existing scrollbar in the window.
- If you want to change any of the `current`, `min`, `max` or `page` values, specify the corresponding parameters. Any of these that you omit won't have their values changed.
- If you want to change the rectangle of a `_scrollOther` scrollbar, specify the new rectangle in the `rect` parameter. If you omit this parameter, the rectangle won't change.
Note: the `rect` parameter is ignored when you're altering a `_scrollVert` or `_scrollHorz` scrollbar.
- You can't alter the type of an existing scrollbar. This parameter is ignored if the scrollbar already exists.

To Activate or De-activate a Scrollbar:

You can use the `Button` statement to activate (highlight) or de-activate (dim) an existing scrollbar.

- To activate it, use: `Button scrollbarID, _activeBtn`
- To de-activate it, use: `Button scrollbarID, _grayBtn`

Linking a Scrollbar's Action to an Edit Field

If you specify a negative `idExpr` value when creating a new scrollbar, `FB^3` looks for a multisyled edit field (which can be static) whose ID number equals `Abs(idExpr)` in the current window. If such a field is found, the behavior of the scrollbar and the edit field become linked, as follows:

- When the user (or your program) moves the scrollbar thumb, the text in the edit field scrolls vertically;
- When the user (or your program) alters the text in the field, or drags through it vertically, the scrollbar thumb moves correspondingly.

When linking a scrollbar to an edit field, it's recommended that you use a vertical scrollbar. A horizontal scrollbar will not scroll the text horizontally, and its action is likely to look strange to the user.

You can't alter the `current`, `min` nor `max` parameters of a scrollbar that's linked to an edit field. These values are automatically altered dynamically as the content and position of the text in the field changes. Also, you can't use the `Button` statement to activate nor de-activate this kind of scrollbar; the scrollbar is active when the edit field contains more lines than will fit in the view rectangle, and is inactive otherwise.

Note: To link a scrollbar to an edit field, you must create the edit field first, and the scrollbar second.

Note: You can only link a scrollbar to a multistyled edit field (a field which was created using a negative `idExpr` value).

Using the Scrollbar

To make the scrollbar useable, your program must call `HandleEvents` periodically. Among other things, `HandleEvents` tracks the motion and clicking of the mouse in the scrollbar, and moves the thumb in response to these user actions. Your program can also move the thumb explicitly by setting the `current` parameter in the `Scroll Button` statement.

Whenever the user moves the thumb, a dialog event of type `_btnClick` is generated. The "id" value for this event equals the scrollbar's ID. You can determine the thumb's current position using the `Button` function:

```
thumbPosition = Button(scrollBarID)
```


Note:

To remove a scrollbar, use the `Button Close` statement:

```
Button Close scrollBarID
```

To find out information about a scrollbar, use the `Button&` function to get the scrollbar's control record.

See Also:

`Button&`; `Button function`; `Button statement`; `Edit Field`; `Dialog function`

Segment	statement	
<i>✗ Appearance</i>	<i>✗ Standard</i>	<i>✗ Console</i>

Syntax:

```
Segment [minSegmentSize]
```

Description:

This statement does nothing in FB^3. It is retained for compatibility with older versions of FutureBASIC.

Segment Return		statement
<i>X Appearance</i>	<i>X Standard</i>	<i>X Console</i>

Syntax:

`Segment Return`

Description:

This statement does nothing in FB^3. It is retained for compatibility with older versions of FutureBASIC.

Select Case

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax 1:

```

Select [Case] targetExpr
    Case testExpr [, testExpr ...] [, max32TestExpr ...]
        [statementBlock]
    [Case testExpr [, testExpr ...] [, max32TestExpr ...]
        [statementBlock]...
    [Case Else
        [statementBlock]]
End Select

```

Syntax 2:

```

Select [Case]
    Case booleanExpr [, booleanExpr ...] [, max32T booleanExpr ...]
        [statementBlock]
    [Case booleanExpr [, booleanExpr ...] [, max32T booleanExpr ...]
        [statementBlock]...
    [Case Else
        [statementBlock]]
End Select

```

Description:

The `Select Case` statement marks the beginning of a “select block,” which must end with an `End Select` statement. You can use a select block to conditionally execute a block of statements based on a number of tests that you specify. When there is only one test, it’s often more convenient to use a `Long If...[Xelse]...End If` block. A select block is useful when there are two or more conditions to test.

If you use Syntax 1, *targetExpr* must be a numeric or string expression. Each *testExpr* has the following syntax:

```
[=|<|>|<=|>|=] expr
```

where *expr* is an expression of the same type as *targetExpr*. When the select block executes, *targetExpr* is compared against each *testExpr* in order. If the *testExpr* does not include a comparison operator (<, >, etc.), then *targetExpr* is compared for equality with *expr*; otherwise *targetExpr* is compared with *expr* using the indicated operator.

If the comparison of *targetExpr* with *expr* is true, the *statementBlock* (if any) following that `Case` statement is executed; then execution continues at the first statement after `End Select`. If none of the comparisons in any `Case` statement is true, the *statementBlock* (if any) following the `Case Else` statement is executed; then execution continues at the first statement after `End Select`. Each *statementBlock* can consist of any number of executable statements, possibly including other `Select...End Select` blocks.

If you use Syntax 2, each *booleanExpr* must be a numeric expression that can be evaluated as “true” (nonzero) or “false” (zero). Typically, *booleanExpr* will be an expression that includes operators such as And, Or, >, <, etc. When the select block executes, each *booleanExpr* is evaluated in order, until one is found that is nonzero (“true”). Then the *statementBlock* (if any) under the corresponding Case statement is executed; then execution continues at the first statement after End Select. If every *booleanExpr* is zero (“false”), the *statementBlock* (if any) following the Case Else statement is executed; then execution continues at the first statement after End Select.

Note:

Remember that the two syntaxes work differently. It’s a common mistake to do something like this:

```
Select Case myVar%
  Case myVar% = 3
    :
  Case myVar% = 7
    :
  Case Else
    :
End Select
```

Here the programmer probably intended to compare the value of *myVar%* to the values 3 and 7; but that’s not what this select block does. Instead, it compares the value of *myVar%* first to the value of “*myVar%=3*” (which is either –1 or 0), and then to the value of “*myVar%=7*” (which is either –1 or 0). This block should have been written in one of these ways:

Using Syntax 1:

```
Select Case myVar%
  Case 3
    :
  Case 7
    :
  Case Else
    :
End Select
```

Using Syntax 2:

```
Select Case
  Case myVar% = 3
    :
  Case myVar% = 7
    :
  Case Else
    :
End Select
```

See Also:

Long If; On <expr> Gosub; On <expr> Goto

SendAppleEvent

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
SendAppleEvent eventtype&, eventClass&, dataAddress&,-  
               dataSize&, processName$
```

Revision:

June, 2001 (Release 5)

Description:

This statement is used in conjunction with other Apple Event commands available starting with Release 5. It allows you to send information of any size to another process. The parameters include the standard event type and class. (See `On AppleEvent` for details about these parameters.) The `dataAddress&` specifies where the Apple Event Manager will find the data and the `dataSize&` variable tells the length of that data.

The `processName$` parameter may be a null string (to send the information to all running processes) or it may be a specific process name. (See `GetProcessInfo` for an example of how to build a list of running processes.)

See Also:

```
HandleEvents, On AppleEvent, GetProcessInfo, Kill AppleEvent;  
AppleEventMessage$
```

SetSelect

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
SetSelect startSelect%, endSelect%
```

Description:

If the current output window contains an active edit field, this statement selects (highlights) a range of text in the field, or sets the position of the blinking insertion point. It also scrolls the selected text into view, if it's not already in view.

The *startSelect%* and *endSelect%* parameters refer to positions between characters in the field. Position 0 is just to the left of the first character; position 1 is between the first and second characters; and so on. If you specify a position greater than or equal to the number of characters in the field, it indicates a position just to the right of the last character. If *startSelect%* equals *endSelect%*, then no text is highlighted, but a blinking insertion point is placed at the indicated position.

Example:

If you specify `SetSelect 0,0`, a blinking insertion point is placed at the beginning of the field's text. If you specify `SetSelect 0,32767`, all of the text in the field is selected. If you specify `SetSelect 32767,32767`, a blinking insertion point is placed at the end of the field's text.

The following inserts or replaces the selected range with the contents of *newString\$*:

```
SetSelect startSelect%, endSelect%
TEKey$ = newString$
```

Note:

Text selection and insertion-point placement are normally handled automatically by the `HandleEvents` statement, in response to the user's mouse and keyboard actions. Use `SetSelect` for special situations.

Use the `Window(_selStart)` and `Window(_selEnd)` functions to determine the active field's current selection range.

Console behavior:

When you use the Console runtime, `SetSelect` switches to the Text Window, then highlights the indicated range of characters in the window.

See Also:

`Edit Field`; `TEKey$ statement`; `Window function`; `TEHandle`

SetZoom**statement**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```

SetZoom [#]WindowID [, (h1,v1)-(h2,v2) ]
SetZoom [#]WindowID, @rect

```

Revision:

June 12, 2000 (Release 3)

Description:

When you create a window that has a zoom box, FB³ assigns a default “zoom rectangle” for the window. The zoom rectangle is the screen rectangle that the window occupies after the user clicks the zoom box. Clicking the zoom box a second time causes the window to change back to its previous rectangle. By default, the zoom rectangle takes up most of the screen.

Use the `SetZoom` statement to assign a new zoom rectangle for the window, specified by $(h1,v1)-(h2,v2)$, where $(h1,v1)$ and $(h2,v2)$ are diagonally opposite corners of the rectangle, in global screen coordinates. The next time the user clicks on the window’s zoom box, and your program subsequently executes `HandleEvents`, the window’s size and location will change to fit the specified zoom rectangle.

If you omit the $(h1,v1)-(h2,v2)$ parameter, `SetZoom` resets the zoom rectangle to the default zoom rectangle.

`SetZoom` has no effect on windows that don’t have a zoom box.

Note:

You can use the `Dialog` function to determine when the user has clicked on the window’s zoom box.

The size of the zoom rectangle is not limited by the `MaxWindow` or `MinWindow` statement.

Do not use `SetZoom` unless the window specified by `WindowID` has already been created (using the `Window` statement).

See Also:

`MaxWindow`; `MinWindow`; `Dialog` function; `Window` statement

Sgn

function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
signOfExpr = Sgn(expr)
```

Description:

Use this function to determine the “sign” of *expr*. *Sgn* returns:

- 1 if *expr* is positive;
- 0 if *expr* is zero;
- -1 if *expr* is negative.

Example:

This `For` loop counts up if *first*<*last*, and counts down if *first*>*last*:

```
For x = first To last Step Sgn(last-first)  
  Print x  
Next
```

See Also:

`For`; `Abs`

ShutDown

statement✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**`ShutDown [msg$]`**Description:**

When used without the `msg$` parameter, `ShutDown` behaves identically as the `End` statement. If the `msg$` parameter is included, the string in `msg$` is displayed in an alert box before the program quits.

Console behavior:

When you use the Console runtime, the string in the `msg$` parameter is displayed in the Text Window, rather than in an alert box.

See Also:`End; System`

Sin**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
theSine# = Sin(expr)
```

Description:

Returns the sine of *expr*, where *expr* is given in radians. The returned value will be in the range -1 to +1. Sin always returns a double-precision result.

Note:

To find the sine of an angle *degAngle* which is given in degrees, use the following:

```
theSine# = Sin(degAngle * pi# / 180)
```

where *pi#* equals 3.14159265359

See Also:

```
Asin; Cos; Tan; Usr _sine
```

Sinh

function

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**`result# = Sinh(expr)`**Description:**

Returns the hyperbolic sine of *expr*. Sinh always returns a double-precision result.

Note:

Sinh(x) is defined as: $\frac{e^x - e^{-x}}{2}$

See Also:

Asinh; Cosh; Tanh; Exp

SizeOf

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
dataSize = SizeOf ({var|typeName|ptrType^|hdlType^^})
```

Description:

This function returns the number of bytes of memory allocated for a particular variable *var*, or the number of bytes allocated for each variable of a particular specified type.

If you specify *typeName*, it should either be the name of a type defined previously in your program (in a `Begin Record` statement or a `#Define` statement), or the name of one of FutureBASIC's built-in types (such as `Int`, `Long`, `Rect`, etc.). `SizeOf` returns the size of a variable of that type.

If you specify *ptrType^*, then *ptrType* should be the name of a type which was previously declared to be a pointer to some other type (in a `#Define` statement). In this case, `SizeOf` returns the size of the type that *ptrType* points to. Note that if you omit the “^” symbol, `SizeOf(ptrType)` just returns the size of a pointer variable (usually 4).

If you specify *hdlType^^*, then *hdlType* should be the name of a type which was previously declared to be a handle to some other type (in a `#Define` statement). In this case, `SizeOf` returns the size of the type referenced by *hdlType*. Note that if you omit the “^ ^” symbols, `SizeOf(hdlType)` just returns the size of a handle variable (usually 4).

Note:

`SizeOf(stringVar$)` returns the number of bytes reserved in memory for the string variable *stringVar\$*. This is not the same thing as `Len(stringVar$)`.

If a variable *HandleVar* contains the handle to a relocatable block (of a possibly unknown type), you can use the Toolbox function `GetHandleSize` to determine the size of the block.

See Also:

`TypeInfo`; `Len`; `Begin Record`; `#Define`

Sound End

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

Sound End

Description:

This statement stops the sound that was initiated with the latest Sound <frequency> or Sound <snd> statement. If the sound is an “snd ” resource that you played using the Sound *resName\$* syntax or the Sound %*resID*% syntax, the Sound End statement also releases the resource.

See Also:

Sound <frequency>; Sound <snd>; Sound%

Sound <frequency>**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Sound pitch,duration [, [volume] [, async]]
```

Description:

This statement plays a tone of the given pitch, duration and volume.

pitch This parameter is expressed as a frequency in cycles per second. The frequency that you specify is converted to the nearest “MIDI note value,” which determines the actual note that’s played. Middle “C” corresponds to a frequency of 261.625.

duration This parameter is expressed in ticks, and can range from 0 to 32,767. However, the toolbox sound commands require FB^3 to translate the ticks into an integer to represent half-milliseconds. This means you can play a note that is no longer than 32.8 seconds.

volume This parameter can range from 0 through 127. Specifying 0 will result in silence, and 127 will play the sound at the maximum volume specified in the “Sound” control panel. If you omit this parameter, it is treated as 127.

async If *async* is `_zTrue`, the sound will play asynchronously. If *async* is `_false`, or you omit the parameter, the sound plays synchronously. When you play asynchronously, your program starts executing the next statement immediately while the sound is playing in the background. When you play synchronously, the next statement in your program does not execute until the sound has finished playing.

If you are playing notes asynchronously, and your execute a second `Sound` statement while another sound is still playing in the background, the new sound won’t start playing until the first sound finishes. Note that on some machines, this technique can result in lost sounds. When playing asynchronously it’s better to use the `Sound%` function to determine when one sound has ended, before attempting to play the next sound.

One way to play sound frequencies is to use negative numbers (from -1 through -127) to represent the note that you wish to play. The table below shows how to use these values.

	A	A#	B	C	C#	D	D#	E	F	F#	G	G#
Octave 1				0	1	2	3	4	5	6	7	8
Octave 2	9	10	11	12	13	14	15	16	17	18	19	20
Octave 3	21	22	23	24	25	26	27	28	29	30	31	32
Octave 4	33	34	35	36	37	38	39	40	41	42	43	44
Octave 5	45	46	47	48	49	50	51	52	53	54	55	56
Octave 6	57	58	59	60	61	62	63	64	65	66	67	68
Octave 7	69	70	71	72	73	74	75	76	77	78	79	80
Octave 8	81	82	83	84	85	86	87	88	89	90	91	92
Octave 9	93	94	95	96	97	98	99	100	101	102	103	104
Octave 10	105	106	107	108	109	110	111	112	113	114	115	116
Octave 11	117	118	119	120	121	122	123	124	125	126	127	128

Using the midi table for a guideline, we can create a version of "pop goes the weasel" as follows:

```
Print "Pop! "; : Sound -70, 45 ,,_false
Print "goes "; : Sound -64, 30 ,,_false
Print "the "; : Sound -67, 15 ,,_false
Print "wea"; : Sound -66, 40 ,,_false
Print "sel "; : Sound -62, 45 ,,_false
```

See Also:

```
Sound%; Sound End; Sound <snd>
```


Sound <snd>**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:****Sound** *soundIDstring\$***Sound** *%resIDNumber***Sound** *&soundHandle&***Description:**

This statement plays a synthesized sound which is in the “snd ” format. The *soundIDstring\$* parameter identifies the sound you want to play; you can construct this string in any of the following ways:

- By resource name: Set *soundIDstring\$* to the name of an “snd ” resource in a currently open resource file. Don’t use a resource name that begins with “%” or with “&”, or it will be confused with the other forms discussed below.
- By resource ID number: If *resID%* is the resource ID number of an “snd ” resource in a currently open resource file, you can prefix the number with a percent sign.

```
Sound %myResID
```

- By “snd ” handle: If *sndH&* is a handle to a sound in “snd ” format, you can construct the *soundIDstring\$* parameter as follows:

```
Sound &mySoundHandle&
```

If FB^3 needs to load a resource to play the sound, it does not automatically release the resource after the sound stops. In this case, you should either use purgeable “snd ” resources, or you should execute the **Sound End** statement to force the resource to be released.

Sounds played using the **Sound <snd>** statement are always played asynchronously; that is, your program continues to the next statement immediately, while the sound is playing in the background. However, if you execute a second **Sound <snd>** statement while a previous sound is still playing in the background, the new sound won’t start playing until the first sound finishes.

Example:

If you want your program to “wait” until a sound finishes before continuing, you can use the **Sound%** function:

```
Sound "reallyLongSound"
While Sound%
    ' (stays in this loop until sound finishes)
Wend
```

See Also:

```
Sound%; Sound <frequency>; Sound End
```

Sound%**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
soundIsPlaying = Sound%
```

Description:

This function returns `_zTrue` if a sound is currently playing, or `_false` otherwise. This applies to sounds that you play using the `Sound <frequency>` statement or the `Sound <snd>` statement, but does not apply to Text-to-speech sounds.

Example:

The `Sound%` function is useful for determining when an asynchronous sound has finished playing.

```
Sound "Quack"
startTime& = Fn TickCount
While Sound%
Wend
endTime& = Fn TickCount
Print "That sound lasted"; endTime& - startTime&; "ticks."
```

Note:

To determine whether a Text-to-speech sound is currently playing, use the Toolbox function `Fn SpeechBusy`.

See Also:

```
Sound <frequency>; Sound <snd>; Sound End
```

Space\$

function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
stringOfSpaces$ = Space$(numChars)
```

Description:

Returns a string consisting of *numChars* space characters. *numChars* must be in the range 0 through 255. Space\$(0) returns an empty (zero-length) string.

See Also:

Print; String\$

Spc

function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

Spc (*numChars*)

Description:

When used with `Print` or `LPrint`, this outputs the number of spaces specified by *numChars*.

Example:

```
Print "Hello" Spc(10) "out there."
```

Program output:

```
Hello          out there.
```

See Also:

`Print`, `LPrint`; `String$`

Sqr

function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

`squareRoot# = Sqr(expr)`

Description:

Returns the square root of *expr*. *Sqr* always returns a double-precision result.

See Also:

Usr _sqroot

Stop

statement

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
Stop [string$ ]
```

Description:

This statement interrupts program execution and displays a dialog box that shows the line number where the `Stop` statement appeared, and gives the user the option of continuing or stopping the program. If the optional string parameter is used, it will be displayed as a message in the alert. If no `string$` parameter is used, FB^3 displays the name of the file in which the `Stop` statement was used.



If the user clicks the “Stop” button, FB^3 calls your stop-event handling routine (if you’ve designated one using the `On Stop` statement), and then your program ends. If the user clicks the “Continue” button, execution continues at the next statement following the `Stop` statement.

`Stop` is mainly useful as a debugging tool. It’s not recommended for use in the “production” version of your program.

Console behavior:

When you use the Console runtime, the `Stop` statement does not display a dialog, but instead prints a message in the Text Window indicating the line number where the `Stop` occurred. At that point, the program ends; the user does not have the option to continue.

See Also:

`End;` `On Break;` `On Stop`

Str#**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
stringFromList$ = Str#(resourceID%, index%)
```

Description:

This function returns a string element from a resource of type “STR#”. The *resourceID%* should specify the resource ID number of an “STR#” resource in a currently open resource file. *index%* indicates which string element to get; the first element is numbered 1. If the “STR#” resource isn’t found, or if *index%* is greater than the number of strings in the resource, the **Str#** function returns an empty (zero-length) string.

“STR#” resources hold lists of strings. Among other things, they’re useful in helping you to localize your program so that it supports the native language of your user. You typically use a program like ResEdit to create an “STR#” resource and its strings; you can also add strings to an “STR#” resource using the **Def ApndStr** statement, or remove them using the **Def RemoveStr** statement.

Note:

You can use the following function to determine how many strings are in an “STR#” resource.

```
Local Fn GetSTRcount(resID%)
  resHndl& = Fn GetResource("STR#",resID%)
  Long If resHndl&
    resCount% = {[resHndl&]}      'Get 1st word in block
  Xelse
    resCount% = 0                 'Couldn't get the resource
  End If
End Fn = resCount%
```

See Also:

```
Def ApndStr; Def RemoveStr; Compile _strResource; Str&
```

Str\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
numberString$ = Str$(numericExpr)
```

Description:

This function returns a string consisting of the characters in the decimal representation of *numericExpr*. If the number is non-negative, then *numberString\$* will also have a leading space character (if the number is negative, the first character will be the minus sign). *Str\$* formats the number in a reasonable way; if you need the number to appear in a special format, use string functions such as *Using*, *Hex\$*, etc.

Generally speaking, *Str\$* is the inverse of the *Val* function.

See Also:

```
Val; Print; Hex$; Using
```


Str& function

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
stringFromHandle$ = Str&(Handle&, index%)
```

Revision:

June, 2001 (Release 5)

Description:

This function returns a string element from a handle that is in the same format as a resource of type "STR#". The *Handle&* should specify the location of the memory block. *index%* indicates which string element to get; the first element is numbered 1. If the handle isn't found, or if *index%* is greater than the number of strings in the resource, the *Str&* function returns an empty (zero-length) string.

Example:

This example creates and fills a handle in the form of a STR# resource and displays the results..

```
Dim sHndl As Handle
Dim x As Word

// create an empty STR# style handle
sHndl = Fn NewHandleClear(2)

// Fill the handle with ASCII strings (1-10)
For x = 1 To 10
    Def ApndStr("This is number"+Str$(x), sHndl)
Next

// Display the handle using Str&
For x = 1 To 10
    Print Str&(sHndl,x)
Next

// We made it. We must dispose of it.
Def DisposeH(sHndl)
```

See Also:

```
Def ApndStr; Def RemoveStr; Compile _strResource; Str#
```

String\$ & String\$\$

function

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
stringOfChars$ = String$(numChars,{Char$|asciiValue%})
container$$ = String$$(numChars,{Char$|asciiValue%})
```

Revision:

July 27, 2000 (Release 3)

Description:

This function returns a string or a container consisting of *numChars* repetitions of a single character. If you specify a string (*Char\$*) in the second parameter, the first character of *Char\$* is repeated. If you specify a number (*asciiValue%*) in the second parameter, *String\$* repeats the character whose ASCII value is *asciiValue%*. *numChars* must be in the range 0 through 255; if *numChars* equals zero, *String\$* returns an empty (zero-length) string.

Example:

```
Print String$(12, "Log")
Print String$(9, 70)
```

Program output:

```
LLLLLLLLLLLLLL
FFFFFFFFFFFF
```

See Also:

Space\$; Asc; Chr\$; Def BlockFill; Appendix F: *The ASCII Character Codes*

StringList

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

StringList On
StringList Off
StringList
StringList = resID [,ResourceName$]
StringList Opt
StringList Nopt
StringList End

```

Description:

When source code is compiled, FB³ needs to know where to store quoted strings. (Quoted strings are those that appear in the source code like "Hello".) The strings are placed either in a STR# resource (for easy access during localization) or in blocks of data that are stored as part of the program.

<i>Statement</i>	<i>Description</i>
• StringList On	Use this command to begin storing strings in a STR# resource.
• StringList Off	Use this command to stop storing strings in a STR# resource and to place them in a block that becomes part of the compiled application.
• StringList	Use this command to restore the StringList state to whatever it was before a StringList On, StringList Off, or StringList = statement.
• StringList = resID [,ResourceName\$]	After the compiler sees this statement, subsequent strings will be stored in a STR# resource with an ID of <i>resID</i> and, optionally, a name of <i>ResourceName\$</i> .
• StringList Opt	This statement optimizes the string list so that there are no duplicates. It optimizes both resource based (StringList On) and code based (StringList Off) storage. This is the most desirable state and leads to smaller applications. When the compiler sees a string appear for a second time in your source code, it will not make a new entry, but will point to the existing identical entry.
• StringList Nopt	Use this command to turn optimization of string off. In this case, the compiler will store separate entries for every occurrence of a string even if those occurrences are identical.
• StringList End	This statement closes out the addition of strings to the current STR# resource and creates a new STR# (or uses an existing STR#) at the next consecutive ID.

Note:

StringList statements are not nestable.
StringList statements don't apply for string constants.

Swap statement

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Swap variable1, variable2
```

Description:

Swap exchanges the contents of the two indicated variables. Both variables must be of the same type.

Example:

```
varOne% = 1200
varTwo% = 999
Print varOne%, varTwo%
Swap varOne%, varTwo%
Print varOne%, varTwo%
```

Program output:

```
1200      999
999      1200
```

See Also:

Let; *Appendix C:Data Types and Data Representation*

SysError

function

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
systemErrorCode% = SysError
```

Description:

This function returns the value of the `SysError` variable, which is an internal variable that FB^3 uses to store the MacOS result codes from disk i/o operations.

While `SysError` usually returns toolbox result codes, it also returns three error codes that are unique to FB. These are:

- 9999 The file number used was outside of the legal range.
Example:
 Open "I", #-1000
- 8888 The program attempted to open a file using a file number that was already in use.
Example:
 Open "R", #1, "Test"
 Open "R", #1, "Something Else"
- 6666 An illegal open type was used.
Example:
 Open "XYZ", #1, "Test"

See Also:

```
SysError statement; Error function; On Error
```

SysError statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
SysError = OSErrorCode%
```

Description:

This statement sets the value of FutureBASIC's internal `SysError` variable. FB^3 sets this variable automatically when you perform disk file operations. The `SysError` statement can be used to reset the variable to zero after one of these other operations has set it to a nonzero value. In virtually every case, you will use `_noErr` (0) as the *OSErrorCode%* parameter.

Example:

```
myHandle& = Fn NewHandle(2000000000)
Long If SysError = _memFullErr
  Print "Out of memory!"
  SysError = _noErr      'Reset error condition
End If
```

Note:

The internal `SysError` variable does not behave like an ordinary variable. For example, you cannot set its value by means of an `Input` or `Read` statement.

See Also:

`SysError` function; `Error` function; `On Error`

System

function

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
systemInFormation = System(whichInfo)
```

Revision:

February, 2002 (Release 6)

Description:

This function returns various kinds of information about the system and about the current application. Set *whichInfo* to one of the values shown here:

<i>whichInfo</i>	<i>Value</i>	<i>System(whichInfo) returns :</i>
<code>_lastCurs</code>	0	“CURS” resource ID number of the current cursor (if it is a resource).
<code>_aplVol</code>	1	A working directory reference number for the folder that contains the application file. This is useful in case you want to read or write files in the same folder as the application.
<code>_sysVol</code>	2	A working directory reference number for the System folder. (Note: to access special folders within the System folder, the preferred method is to use the Toolbox function <code>FindFolder</code> .)
<code>_macPlus</code>	3	<code>System(_macPlus)</code> always returns 0. This is retained for compatibility with older versions of FutureBASIC.
<code>_aplRes</code>	4	Reference number for the Application file’s resource fork (which is always open while your application is running).
<code>_memAvail</code>	5	Number of free bytes currently available in the application’s heap. Returns the same value as <code>Mem(_freeBytes)</code> .
<code>_scrnWidth</code>	6	The width of the main monitor, in pixels.
<code>_scrnHeight</code>	7	The height of the main monitor, in pixels.
<code>_sysVers</code>	8	An integer representing the current System version number. For example, 761 represents version 7.6.1.

<code>_aplActive</code>	9	Returns a positive value if the application is the currently active (foreground) process. Returns a negative value if the application is currently in the background. Note that FutureBASIC3 also generates a dialog event every time your application is moved to the foreground or to the background (see the <code>Dialog</code> function).
<code>_maxColors</code>	10	Maximum color bit-depth available on the main monitor. Use the expression <code>(2 ^ System(_maxColors))</code> to get the actual number of available colors.
<code>_crntDepth</code>	11	Current color bit depth on the main monitor. Use the expression <code>(2 ^ System(_crntDepth))</code> to get the actual number of available colors.
<code>_cpuType</code>	12	A code indicating the machine's CPU type, for 68k CPU's. Return values: <code>_env68000</code> (1); <code>_env68010</code> (2); <code>_env68020</code> (3); <code>_env68030</code> (4); <code>_env68040</code> (5). Note: this value is unreliable for PPC CPU's: to determine
<code>_machType</code>	13	A code indicating the machine type. Use <code>machName\$ = Str#(-16395, System(_machType))</code> to determine the machine's name.
<code>_aplFlag</code>	14	<code>System(_aplFlag)</code> returns <code>_false</code> if the project was run and <code>_zTrue</code> if the project was built.
<code>_cpuSpeed</code> (PPC only)	15	<code>System(_cpuSpeed)</code> returns the current clock speed (in megahertz) of the microprocessor.
<code>_clockSpeed</code> (PPC only)	16	<code>System(_clockSpeed)</code> returns the Gestalt clock speed (in megahertz) of the microprocessor.
<code>_lastCursType</code> (Appearance Manager)	17	This function may return 0 (plain), <code>_themeCursorStatic</code> or <code>_themeCursorAnimate</code> .
<code>_aplvRefNum</code> (Appearance Manager)	18	This is the volume reference number of the running application.
<code>_aplparID</code> (Appearance Manager)	19	This is the volume parent ID number of the running application.

Note:

You can get many other kinds of system-wide information by using the Toolbox function `Fn Gestalt`.

See Also:

The “Gestalt Manager” chapter in *Inside Macintosh: Operating System Utilities*

System	statement	
✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>

Syntax:

System

Description:

This is a synonym for the End statement

Tan

function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
theTangent# = Tan(expr)
```

Description:

Returns the tangent of *expr*, where *expr* is given in radians. **Tan** always returns a double-precision result.

Note:

To find the tangent of an angle *degAngle* which is given in degrees, use the following:

```
theTangent# = Tan(degAngle * pi# / 180)
```

where *pi#* equals 3.14159265359.

See Also:

Atn; Sin; Cos

Tanh

function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

`result# = Tanh(expr)`

Description:

Returns the hyperbolic tangent of *expr* where *expr* is given in radians. Tanh always returns a double-precision result.

Note:

Tanh (x) is defined as: $\frac{e^x - e^{-x}}{e^x + e^{-x}}$

See Also:

Asinh; Cosh; Sinh; Tan; Exp

TBAlias**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:****TBAlias** *oldName*, *NewName***Description:**

As the Mac OS matures, terse and economical toolbox names give way to more descriptive names. In C, this spells disaster because there is no way to stay in step except to go through all of your projects and change the name of the toolbox call to match Apple's latest syntax. FB^3 offers a more elegant solution by providing an alias. A toolbox alias is a synonym that FB^3 uses to translate all of your old syntax into the latest PPC toolbox names.

This happens more often than you might imagine. Some common aliases are shown below:

```
TBAlias InsertMenuItem ,InsMenuItem
TBAlias DeleteMenuItem ,DelMenuItem
TBAlias GetMenuHandle ,GetMHandle
TBAlias CountMenuItems ,CountMItems
TBAlias GetControlValue ,GetCtlValue
```

Without these TBAlias statements, your existing code would break when it used a common toolbox like GetMHandle. With the alias in place, FB^3 sees "GetMHandle" and automatically translates it to "GetMenuHandle". The result is that your old code continues to work in a modernized environment.

See Also:

Library; Toolbox

TEHandle**function**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
editFieldHandle& = TEHandle(fieldID)
fieldInfo = TEHandle(-fieldID)
```

Description:

This function returns information about an edit field or picture field in the current output window.

If you use the first syntax (passing a positive value to `TEHandle`), `TEHandle` returns a handle to the TextEdit record associated with the specified field. Although a valid handle will also be returned for picture fields, this is really more useful in the case of edit fields.

The following table lists some of the more useful elements that can be found in the field's TextEdit record. The syntax shown in the table assumes that the value returned by `TEHandle` has been assigned to a long integer variable `teH&`.

<i>Record element</i>	<i>Description</i>
teH&..teViewRect.top% teH&..teViewRect.left% teH&..teViewRect.bottom% teH&..teViewRect.right%	Rectangle (in local window coordinates) where the text is visible).
teH&..teJust%	Text justification: _teJustCenter = 1; _teJustRight = -1; _teJustLeft = 0.
teH&..teLength%	Total number of characters in the field.
teH&..teTextH&	Handle to the field's text.
teH&..teNLines%	Number of lines of text in the field.
[teH&] + _teLines	Address of the beginning of the "lineStarts" array, which tells the character position associated with the beginning of each line.

(Note: for a description of other elements in the TextEdit record, see the "TextEdit" chapter in Inside Macintosh: *Text*.)

If you use the second syntax (passing a negative value to `TEHandle`), `TEHandle` returns a value identifying the field's type, as follows:

<i>TEHandle(-fieldID)</i>	<i>Description</i>
0	No field with ID <i>fieldID</i> exists in this window.
+1	The field is an editable text field.
-1	The field is a static text field.
+2	The field is a clickable picture field.
-2	The field is a non-clickable picture field

Example:

This function returns the next available ID number that is not currently being used by any picture field or edit field in the current window.

```

Local Fn GetUnusedFieldID
  Dim id
  id = 0
  Do
    Inc(id)
  Until TEHandle(-id) = 0
End Fn = id

```

Note:

You should generally not alter the contents of a TextEdit record directly. Use TextEdit Toolbox routines, or FB^3 statements like `Edit Field`, to alter the characteristics of the field.

If you use `TEHandle` to get a TextEdit record handle, you should not dispose of the handle. The memory associated with the edit field is automatically disposed when you close the edit field (using `Edit Field Close`) or close its window (using `Window Close`).

See Also:

`Edit Field`; `Picture Field`; `Window(_efHandle)`; Inside Macintosh: *Text*

TEKey\$

function

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Char$ = TEKey$
```

Description:

If there is an active edit field in the current window, and you have designated an edit-event handling routine using the `On Edit` statement, `TEKey$` returns a 1-character string indicating a key that the user pressed. You should check the value of `TEKey$` within your edit-event handling routine, each time that routine is called.

When you use an edit-event handling routine, the user’s keystrokes are not transmitted directly to the edit field. Based on the value returned by `TEKey$`, your edit-event handling routine can decide what to do with the keypress; it can transmit it to the field unaltered, or transmit a different character, or perform some other action. Your routine can use the `TEKey$` statement to transmit the keypress (or some other character) to the edit field, if desired. `TEKey$` values that correspond to the backspace key, and the four arrow keys should generally be transmitted to the field unaltered.

Note that any keypresses that can’t alter the contents of the field nor move the insertion point are generally not reported to your edit-event handling routine. Keypresses that will not be reported include the following:

- Arrow keys, if the insertion point is already as far as it can go in the indicated direction (use the `Dialog` function (event types `_efLeftArrow`, `_efRightArrow`, `_efUpArrow` and `_efDownArrow`) to detect arrow keys in this situation);
- Command-key equivalents for menu items;
- ⌘-period.
- The “Tab” key (use the `Dialog` function (event type `_efTab` or `_efShiftTab`));
- The “Clear” key (use `Dialog` function (event type `_efClear`));
- The “Return” key, if the field is one of the “NoCR” types (use the `Dialog` function (event type `_efReturn`));

See Also:

```
On Edit; Dialog function; TEKey$ statement
```


TEKey\$

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
TEKey$ = string$
```

Description:

If there is an active edit field in the current window, *string\$* is inserted into the field at the current insertion point, or replaces the currently highlighted text in the field. The insertion point is then placed at the end of the newly-inserted text, and the text is scrolled into view if it is not already in view.

See Also:

```
TEKey$ function; Edit Field; On Edit
```

Text**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Text [font%][,[size%][,[face%][,copyMode%]]]
```

Description:

This statement sets the text characteristics for the current output window or printer. It affects subsequent `Print` statements (in the current window or printer), subsequent `LPrint` statements, and subsequently-created edit fields (in the current window). It does not change the appearance of any existing text, except in buttons created using the `_useWFont` type modifier (see the `Button` statement).

Each of the parameters is optional. If you omit a parameter, the corresponding characteristic won't be changed. The parameters are interpreted as follows:

- *font%* A number which identifies the font family. Certain common fonts have standard numbers which are identified by these constants:

<code>_newYork</code>	<code>_venice</code>	<code>_geneva</code>	<code>_monaco</code>
<code>_times</code>	<code>_helvetica</code>	<code>_courier</code>	<code>_symbol</code>

The following constants are also available:

<code>_sysFont</code>	(System font; usually Chicago or Charcoal)
<code>_applFont</code>	(Default application font; usually Geneva)

For other fonts, the best way to determine the font number is to pass the font's name to the `GetFNum` Toolbox procedure or to the new Toolbox function called `FMGetFontFamilyFromName` as here:

```
Call GetFNum(fontName$, font%)
font% = Fn FMGetFontFamilyFromName(fontName$)
```

When you do this, the font's number is returned in the *font%* variable, which you can then pass to the `Text` statement.

- *size%* The font size, in points. This usually gives some indication of the height in pixels of the tallest character; however, you shouldn't rely on this. Use the `Usr FontHeight` function to determine the pixel height of a line of text.

- *face%* The text style. To set the face to "plain," set this parameter to zero; otherwise, you can set it to the sum of any of the following bitmask values:

<code>_boldBit%</code>	<code>_ulineBit%</code>	<code>_outlineBit%</code>	<code>_condenseBit%</code>
<code>_italicBit%</code>	<code>_extendBit%</code>	<code>_shadowBit%</code>	

Example:

```
Text _geneva, 12, _boldBit% + _italicBit%
```

- *copyMode%* This determines how the text interacts with the existing background. The most commonly used values are `_srcCopy` (the entire character's rectangle replaces the background), and `_srcOr` (only the character's glyph replaces the background). Other transfer modes include:

<code>_srcXor</code>	<code>_addOver</code>
<code>_srcBic</code>	<code>_subPin</code>
<code>_notSrcCopy</code>	<code>_transparent</code>
<code>_notSrcOr</code>	<code>_adMax</code>
<code>_notSrcXor</code>	<code>_subOver</code>
<code>_notSrcBic</code>	<code>_adMin</code>
<code>_blend</code>	<code>_grayishTextOr</code>
<code>_addPin</code>	<code>_ditherCopy (mask)</code>

See the “QuickDraw Text” chapter in *Inside Macintosh: Text*, and the chapters “QuickDraw Drawing” and “Color QuickDraw” in *Inside Macintosh: Imaging with QuickDraw*, for more information about text transfer modes.

Note:

Each screen window maintains its own text characteristics separately from the others. The `Text` statement affects only the characteristics in the current window.

To change the characteristics of text in existing edit fields, use the `Edit Text` statement.

To change the color of subsequently printed text, use the `Color` or `Long Color` statement.

See Also:

`Edit Text`; `Route`; `Print`; `Button`; `Usr FontHeight`; `Color`; `Long Color`

ThreadBegin

statement✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
ThreadBegin Fn name [, parameter& [, stackMin&]]
```

Revision:

August, 2002 (Release 7)

Description:

What exactly is a thread? It is a function that performs a task. The difference between a threaded function and any other function is that the threaded version runs in the background. Your program (and other running processes) advance without interruption as the threaded task is performed.

There are many reasons for using threaded functions. One that is becoming popular relates to functions that access Internet connections. Since logging in and downloading often take a good bit of time, it makes sense to hand such a task to a threaded function and go on about your business. You might handle other tasks like complex calculations, lengthy sorts, or time-consuming file loads.

When a threaded function is called, it executes without stopping until it is complete — just like any other function. But at some point inside of the function, you yield to other processes by calling `ThreadStatus`. The `ThreadStatus` statement allows other actions to take place and optionally sets a timed delay for the number of ticks that should elapse before your function regains control.

The *parameter&* passed to a thread may be any long integer required by your program. It is for your use. If a parameter is sent, you should accept it in the named local function. For instance, if a *parameter* were used in the example below, we would create the function with `Local Fn myThread(param As Long)` instead of just plain `Local Fn myThread`.

If the *stackMin&* parameter is omitted, FutureBASIC sets a minimum stack space of 131072 bytes (128K). You may experiment with larger or smaller values for your specific needs.

Example:

This simple example creates a thread that prints a string of text, piece by piece, in a `For/Next` loop. The `ThreadStatus` call installs a callback time of 10 ticks which makes the text appear slowly. You can type into the edit field as the threaded text is placed on the screen.

```

Local Fn myThread
  Dim t$,x, abort
  t$ = "Hello. I am a thread. "
  t$ += "I execute in the background."
  For X = 1 To Len(t$)
    Print @(1,1) Left$(t$,x);
    abort = ThreadStatus(10)           //call me in 10 ticks
    If abort != 0 Then Exit Fn
  Next
End Fn

Window 1
Text _sysfont,12,0,0
Print@(1,3)"Type into the Edit Field"
Edit Field 1,"", (8,80)-(200,100)
ThreadBegin Fn myThread
Do
  HandleEvents
Until 0

```

See Also:

`ThreadStatus`; `Timer`; `On Timer`;

ThreadStatus

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
abortBoolean = ThreadStatus [ (ticks&) ]
```

Revision:

August, 2002 (Release 7)

Description:

In a threaded function, it is necessary to tell the Thread Manager when you wish to yield to other running processes. A thread which yields very little time will run faster, but will cause all other operations on the computer to run at a slower rate.

The `ThreadStatus` function returns a `Boolean` result to indicate whether or not the thread should continue operation. One reason that a thread might be asked to cease operation would be if the computer was about to shut down. When you receive a non-zero result from the `ThreadStatus` statement, you should exit the threaded function immediately.

See Also:

`ThreadBegin`; `Timer`; `On Timer`

Time\$

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
currentTime$ = Time$
```

Description:

This function returns the current time of day as an 8-character string in this format:

```
hh:mm:ss
```

where *hh* is a 2-digit hour in 24-hour format ("00" through "23"), *mm* is a 2-digit minutes value ("00" through "59"), and *ss* is a 2-digits seconds value ("00" through "59").

Example:

This program speaks one of three appropriate phrases depending on the time of day.

```
hour = Val(Left$(Time$,2))
Select Case hour
  Case >= 18
    osErr = Fn SpeakString("Good evening.")
  Case >= 12
    osErr = Fn SpeakString("Good afternoon.")
  Case Else
    osErr = Fn SpeakString("Good morning.")
End Select
While Fn SpeechBusy
Wend
```

Note:

To get a time string which is formatted according to the user's "Date & Time" preferences, use the Toolbox procedure `TimeString`. For example:

```
Dim timeStr$, @ t&
Call GetDateTime( t& )
Call TimeString( t&, _false, timeStr$ , _nil )
Print "The time is now "; timeStr$
```

See Also:

Date\$; Timer function

Timer

function

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
secondsSinceMidnight& = Timer
```

Description:

This function returns the current time of day expressed as a number of seconds since midnight. Its value ranges from 0 through 86399.

Note:

The `Timer` function is useful for measuring elapsed time intervals, as long as you make appropriate adjustments in case the measured interval crosses the midnight boundary. To measure time intervals with a finer resolution than 1 second, use the Toolbox routines `TickCount` or `MicroSeconds`. The `TickCount` function measures the number of “ticks” since startup (there are approximately 60.15 ticks per second), and the `MicroSeconds` procedure measures the number of microseconds since startup.

See Also:

`Time$`

Timer**statement**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```

Timer = interval
Timer(0) Off
Timer(0) End

```

Revision:

January 14, 2001 (Release 4)

Description:

If you have designated a timer-event handling routine (using the `On Timer` statement), the `Timer` statement alters the interval at which timer events occur. If timer events have not yet been initiated (because your `On Timer` statement specified an interval of zero), then the `Timer` statement also initiates timer events.

If the *interval* parameter is greater than zero, it specifies the interval in seconds. If *interval* is less than zero, then `Abs(interval)` specifies the interval in ticks (a tick is approximately 1/60 second). If you set *interval* to zero, the `Timer` statement is ignored; you cannot use the `Timer` statement to disable timer events once they've been initiated.

Timer events are internally queued by FB^3 until your application is ready to accept them. If you are receiving events once per second and your program is occupied with other tasks for 10 seconds, you will receive 10 timer events at the first available opportunity.

If you wish to flush the timer event queue, use `Timer(0) Off` or `Timer(0) End`.

See Also:

`On Timer`

Toolbox

functions

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Toolbox [Fn] functionName [(arg1 [,arg2...])]↵
                        [= returnParam] [`0x0000,0x0000...]
```

Description:

In earlier versions of FutureBASIC, access to toolbox calls was limited to modification of internal resources or `Local Fns` that accessed the 68K traps through assembly language calls. In FB^3, the source for all toolbox calls is located in editable source code. By convention, these items are stored in the Headers folder (path: FB Extensions/Compiler/Headers) and are give a name that begins with "Tlbx". This is not an absolute requirement. You may create toolbox calls and place them in any project include so long as they are defined before they are called.

Routine Name

The name of a routine is case sensitive. This is the only instance in FB^3 where upper and lower case letters may not be freely exchanged. The reason for this is that PPC code actually uses the text of the routine name in a look up table where older 68K code used a hexadecimal trap address.

When you examine the text of a toolbox call, the name appears to be in upper case. This is because the editor uses an internal look-up table to determine how the name should be formatted. In reality, the text stored in memory looks very different. You can see this difference by placing an apostrophe in front of the text to make it a remark.

```
TOOLBOX FN GETCICON(SHORT) = LONG `0xAA1E
'Toolbox Fn GetCIcon(Short) =Long `0xAA1E
```

In this example, the case of `GetCIcon` is specific and will not work if there is any modification. You may determine what case is required by looking at the C code used by Apple in defining the toolbox.

Toolbox Functions

Functions (as opposed to procedures) return a value. Unlike BASIC, the Mac's toolbox routines only return numeric variables. Common results are `Long`, `Short`, and `Boolean`. The toolbox does not return strings or records.

Toolbox functions begin with `Toolbox Fn`. Procedures omit the "Fn". Functions also provide a size for the return value with syntax like "`= Short`".

```
Toolbox Fn FunctionName(param) = Short
```

Toolbox Procedures

Procedures work much the same as toolbox functions, except that the "Fn" prefix is omitted and the return value is no longer necessary.

```
Toolbox ProcName (param)
```

68K & PPC

Using only a text description, toolbox calls can be created and used in PPC. FB³ makes all of the necessary conversions into assembly language and sets up the proper look-up tables in your application. If you want the calls to work in 68K as well as PPC, then you will need to add the required assembly language to the end of the `Toolbox` statement for the selector and the trap.

```
Toolbox Fn Alert(Short,Long) = Short `0xA985
```

While this may look complex, it is really a modification to something that looks very similar in C code. The following was taken from Apple's headers.

```
Alert      (SInt16 alertID,ModalFilterUPP modalFilter)
           ONEWORDINLINE (0xA985);
```

Note that the upper/lower case of the name in FB³'s toolbox function exactly matches the one from Apple (even though it will be reformatted when used in the editor). The code from `ONEWORDINLINE` was moved to the end of the statement after the back apostrophe (also called a grave symbol). While C requires that you predict the number of hex instructions that follow (`ONEWORDINLINE`, `TWOWORDSINLINE`, etc.) FB³ automatically reads the hex digits up to the end of the line or a remark, whichever comes first.

See Also:

```
TBAlias, Library
```

Tron/Troff**statements**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```

Tron [All]
Tron Break
Troff

```

Description:

These statements affect which parts of your program can be traced with the Debugger tool.

In order to build a program that can be traced in the Debugger, you must create your program as a project (using the Project Manager), and in the Project window you must select with a “bug icon” each file that you want to trace. By default, all the lines in the selected files can then be traced. However, there may be cases in which you want to limit the lines that can be traced, especially because traceable lines make your compiled program’s size larger and take somewhat longer to execute.

Troff cause subsequent lines to be untraceable. This is a non-executable statement, so it affects the lines that appear below it, regardless of their actual order of execution. It overrides previous occurrences of **Tron** [**All**].

Tron [**All**] causes subsequent lines to be traceable. This is the default condition for all lines in your file above the first occurrence of **Troff**. This is a non-executable statement, so it affects the lines that appear below it, regardless of their actual order of execution. It overrides previous occurrences of **Troff**. Note that the **All** keyword is optional.

Tron Break is an executable statement which causes the Debugger window to appear (if it was previously hidden), and causes execution to pause. It has the same effect as setting a breakpoint in the Debugger. **Tron Break** is ignored if it appears within a section of code which is not traceable.

See Also:

Compile Long If; Tron X

Tron X**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**`Tron X`**Description:**

The `Tron X` statement is a debugging tool. It examines the event queue for any recent ⌘-period keypresses by the user. If a ⌘-period keypress is found, `Tron X` displays a dialog giving the user the option to quit the program (this is the same dialog that is displayed by `HandleEvents` when the user pressed ⌘-period and there is no designated break-handling routine).

`Tron X` is handy for manually breaking out of a loop. If your program seems to get “stuck” inside a loop for a long time, try inserting a `Tron X` statement inside the loop. The next time you run your program, you will be able to break out of the loop by pressing ⌘-period.

See Also:`Tron/Troff; HandleEvents; On Break`

TypeOf

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
dataType = TypeOf ({variable | typeName})
```

Description:

When FB^3 compiles your program, it associates a unique integer with each data type that your program uses. The `TypeOf` function returns the integer that's associated with the specified type.

typeName should be the name of a type that was previously defined in a `Begin Record` statement or a `#Define` statement; or the name of one of FutureBASIC's built-in types (such as `Int`, `Long`, etc.).

variable can be the name of any variable. In this case, `TypeOf` returns the type ID number associated with the variable's type. Note that if the variable was not previously declared in a `Dim` statement, and has no type-identifier suffix, `TypeOf` will assume that the variable's type is the default type (which is `Int` unless a `Def<type>` statement applies).

Example:

This program uses `TypeOf` to determine what kind of data a pointer points to.

```
Local Fn DoSomething(@varAddr&, varType)
  Print "The data you passed was: ";
  Select varType
    Case TypeOf(Int)
      Print Peek Word(varAddr&)
    Case TypeOf(Long)
      Print Peek Long(varAddr&)
    Case TypeOf(Str255)
      Print PStr$(varAddr&)
    Case Else
      Print "Unknown"
  End Select
End Fn

myInt% = 1623
Fn DoSomething(myInt%, TypeOf(myInt%))
myLong& = 426193
Fn DoSomething(myLong&, TypeOf(myLong&))
myString$ = "Hello"
Fn DoSomething(myString$, TypeOf(myString$))
End
```

Program output:

```
The data you passed was: 1623
The data you passed was: 426193
The data you passed was: Hello
```

Note:

The integer values returned by `TypeOf` are determined dynamically at compile time. You should not count on `TypeOf (someType)` to return the same value every time your program is compiled.

See Also:

`SizeOf`; [Appendix C: *Data Types and Data Representation*](#)

UCase\$ UCase\$\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
upperCaseString$ = UCase$(string$)
upperCaseContainer$$ = UCase$$ (container$$)
```

Description:

This function returns a copy of *string\$* or *container\$\$* with all its lower-case letters converted to upper-case. This also applies to letters with diacritical marks; so for example the string “mágüey” will be converted to “MÁGÜEY”.

UCase\$ is useful when you want to compare two strings for equality without regard to their letter case. For example, suppose you want all of the following strings to be treated in the same way:

```
STAZ
Staz
staz
sTAz
```

Any pair from the above set will be considered to “match” if they’re compared as follows:

```
If UCase$(str1$) = UCase$(str2$) Then Print "Names match."
```

The comparison of containers requires the use of an additional function. See `Fn FBcompareContainers` for more information.

Note:

To test whether one string is greater than or less than another without regard to letter case, an alternative method is to use the string comparison operators “>” and “<”. See Appendix D: *Numeric Expressions*, for more information.

See Also:

FB CompareContainers; Appendix F: *ASCII Character Codes*

UniversalFn**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
returnVar = UniversalFn(procPtr&,flags[,parm1[,parm2 ...]])
```

Description:

This function executes the routine located at the memory address specified by *procPtr&*, optionally passing it the parameters specified in *parm1*, *parm2* etc., and optionally returns a result. The *flags* parameter contains information about how parameters should be interpreted when they're passed to the routine, and what type of value (if any) should be returned. Unlike the `Call` statement, the `UniversalFn` function can be used to make “mixed mode” calls; that is, `UniversalFn` allows your PPC-compiled program to call 68k-compiled routines, and vice-versa. (Note: if you are making a mixed mode call, then *procPtr&* must be a pointer to a “routine descriptor.” See *Inside Macintosh: PPC System Software* for more information.)

In the current version of FB³, `UniversalFn` can only call “Pascal stack-based” routines; this is the style which is most often supported by the MacOS Toolbox. Before you use `UniversalFn`, you should make sure you know how many parameters (if any) the *procPtr&* routine expects, what the expected data type of each parameter is, and the data type (if any) that the routine will return. Having made this determination, you should then set the *flags* parameter to the sum of one or more of the following constants, as appropriate (note: the *flags* parameter must be a static integer expression; it can't be a variable):

<i>Constant</i>	<i>Description</i>
<code>_rtnNone</code> (0)	Routine does not return a value.
<code>_rtnByte</code> (16)	Routine returns a byte value.
<code>_rtnWord</code> (32)	Routine returns a word (2-byte) value.
<code>_rtnLong</code> (48)	Routine returns a long (4-byte) value.
<code>_p1Byte</code> (64)	Routine expects 1st parameter to be a byte.
<code>_p1Word</code> (128)	Routine expects 1st parameter to be a word.
<code>_p1Long</code> (192)	Routine expects 1st parameter to be a long.
<code>_p2Byte</code> (256)	Routine expects 2nd parameter to be a byte.
<code>_p2Word</code> (512)	Routine expects 2nd parameter to be a word.
<code>_p2Long</code> (768)	Routine expects 2nd parameter to be a long.
<code>_p3Byte</code> (1024)	Routine expects 3rd parameter to be a byte.
<code>_p3Word</code> (2048)	Routine expects 3rd parameter to be a word.
<code>_p3Long</code> (3072)	Routine expects 3rd parameter to be a long.
<code>_p4Byte</code> (4096)	Routine expects 4th parameter to be a byte.
<code>_p4Word</code> (8192)	Routine expects 4th parameter to be a word.
<code>_p4Long</code> (12288)	Routine expects 4th parameter to be a long.
<code>_p5Byte</code> (16384)	Routine expects 5th parameter to be a byte.
<code>_p5Word</code> (32768)	Routine expects 5th parameter to be a word.
<code>_p5Long</code> (49152)	Routine expects 5th parameter to be a long.
through...	
<code>_p13Byte</code> = (&40000000)	Routine expects 13th parameter to be a byte.
<code>_p13Word</code> = (&80000000)	Routine expects 13th parameter to be a word.
<code>_p13Long</code> = (&C0000000)	Routine expects 13th parameter to be a long.

Routine constants are located in the `Tlxb Standard.Incl`. The maximum number of parameters that can be used for a universal function is 13.

Example:

```
_myProcFlags = _rtnWord + _p1Word + _p2Long
x% = UniversalFn(procAddr&, _myProcFlags, 307, fred&)
```

Note:

It's vital that you set up the *flags* parameter properly, so that the stack pointer will be adjusted correctly when the routine is called. An incorrectly adjusted stack pointer can cause your program to crash.

See Also:

`UniversalProc`

UniversalProc

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✓ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
UniversalProc(procPtr&, flags[, parm1[, parm2 ...]])
```

Description:

The `UniversalProc` statement works identically to the `UniversalFn` function, except that it should only be used to call routines which don't return any value. The parameters of `UniversalProc` are interpreted the same way as the parameters of `UniversalFn`.

See Also:

```
UniversalFn
```

UnloadSegment

statement

<i>✗ Appearance</i>	<i>✗ Standard</i>	<i>✗ Console</i>
---------------------	-------------------	------------------

Syntax:

`UnloadSegment "statementLabel"`

Description:

This statement does nothing in FB^3. It is retained for compatibility with older versions of FutureBASIC.

Uns\$**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
digitString$ = Uns$(expr)
```

Description:

This function interprets the internal bit pattern of *expr* as an unsigned integer, then returns a string of decimal digits representing that integer's value. The length of the returned string depends on which of `DefStr Byte`, `DefStr Word` or `DefStr Long` is currently in effect; the returned string may be padded on the left with one or more "0" characters, to make a string of the indicated length.

If *expr* is a positive integer, then the number represented in the returned string will be the same as the value of *expr* (provided that the current `DefStr` mode allows `Uns$` to return sufficient digits).

If *expr* is a negative integer, then its internal bit pattern is different from that of an unsigned integer. In this case, the number represented in the returned string will be:

- *expr* + 28, if `DefStr Byte` is in effect;
- *expr* + 216, if `DefStr Word` is in effect;
- *expr* + 232, if `DefStr Long` is in effect.

Note:

To convert a "signed integer" expression *sexpr* into an "unsigned integer" expression which has the same internal bit pattern, just assign *sexpr* to an unsigned integer variable. For example:

```
myUnsLong&` = mySignedLong&
```

See Also:

`DefStr Byte/Word/Long`; `Hex$`; `Oct$`; `Bin$`; *Appendix C: Data Types and Data Representation*

Until**statement**

See the `Do` statement.

Using**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
string$ = Using format$;expr
```

Description:

This function returns a decimal string representation of the numeric *expr*, formatted according to specifications in *format\$*. The characters in *format\$* are interpreted as follows:

<i>Specifier</i>	<i>Description</i>
.	This represents a decimal point. In the returned string, the “.” is replaced by the currently defined decimal point symbol (which is just a period, if <code>Def Using</code> has never been executed). This specifier also indicates where the integer part of <i>expr</i> should be separated from the fractional part in the returned string. If there is more than one “.” character in <i>format\$</i> , only the first one is interpreted as a decimal point specifier. If <i>format\$</i> does not contain a “.” specifier, then only the integer part of <i>expr</i> will be represented in the returned string. Note that “.” is interpreted somewhat differently if the “^^^” specifier is also used.
#	Each “#” that appears to the left of the “.” specifier is replaced by a digit from the integer part of <i>expr</i> , or (if there are excess “#” specifiers) by a blank space. Each “#” that appears to the right of the “.” specifier is replaced by a digit from the fractional part of <i>expr</i> , or (if there are excess “#” specifiers) by a “0” character. There must be at least enough “#” specifiers to the left of the “.” to represent the integer part of <i>expr</i> . If there are too few “#” specifiers to the right of the “.” to represent the fractional part, the value of <i>expr</i> is rounded to the indicated number of digits. Note that “#” is interpreted somewhat differently if the “^^^” specifier is also used.
*	This is treated the same as the “#” specifier, except that if there are excess “*” specifiers to the left of the “.”, the excess specifiers are replaced by “*” characters rather than by blank spaces.
,	Each occurrence of “,” which appears after the first “#” or “*” is interpreted as a thousands separator. If there are enough digits in <i>expr</i> to fill at least one of the “#” or “*” specifiers to the left of the “,”, then the “,” is replaced by the currently defined thousands-separator symbol (which is just a comma, if <code>Def Using</code> has never been executed); otherwise, the “,” is replaced by a blank space.
\$	If “\$” appears to the left of the leftmost “#” or “*” specifier, it’s replaced by the currently defined currency symbol (which is just a dollar sign, if <code>Def Using</code> has never been executed). However, if this would cause blank spaces to appear between the currency symbol and the number, then the currency symbol is moved to the right until there are no intervening blank spaces.

<i>Specifier</i>	<i>Description</i>
+	This is replaced by a “+” if <i>expr</i> is positive, or by a “-” if <i>expr</i> is negative. As with “\$”, the symbol may be moved to the right to eliminate intervening blank spaces.
-	This is replaced by a blank space if <i>expr</i> is positive, or by a “-” if <i>expr</i> is negative. As with “\$”, the symbol may be moved to the right to eliminate intervening blank spaces. This specifier is most useful if you want the “-” to appear in a non-standard location; if you use neither the “-” nor the “+” specifier, and <i>expr</i> is negative, and there is a sufficient number of “#” symbols used as placeholders, a “-” will always appear to the left of the number.
^^^^	This specifier causes <i>expr</i> to be represented in scientific notation. The “^^^^” characters will be replaced by an exponent expression, in the form “E+nn” or “E-nn”. When you use the “^^^^” specifier, the leftmost “#” specifier in <i>Format\$</i> is always replaced by the first significant (nonzero) digit in <i>expr</i> . “Standard” scientific notation places one nonzero digit before the decimal point; however, you can adjust the “#” and “.” specifiers to place as many digits before and/or after the decimal point as you want; adjusting the number of “#” specifiers left of the decimal point will cause a corresponding adjustment in the exponent number.
^^^^^	This is the same as the “^^^^” specifier, but it allows for up to 3 digits in the exponent. You should use this specifier whenever there’s a chance that the exponent could exceed ± 99 .

If *format\$* contains any characters other than the specifiers listed above, they are transferred unaltered to the returned string.

Example:

```
x! = 14.726
Print Using "You owe me $#,###.##."; x!
```

Program output:

```
You owe me $14.73.
```

See Also:

```
Def Using; Str$; Uns$
```

Usr**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
returnValue& = Usr fnIndicaTor (expr)
```

Description:

This function calls one of several low-level subroutines built into FB^3. The *fnIndicaTor* parameter indicates which routine to call; use one of the constants shown in the table below.

<i>Function</i>	<i>Description</i>
Usr _allocLockBlk(size&)	Allocates a new non-relocatable block which is size& bytes long, clears the block's contents to zero, and returns a pointer to the block. Same as Fn NewPtrClear(size&).
Usr _releaseBlk(Ptr&)	Releases the non-relocatable block pointed to by Ptr&. Same as Call DisposePtr(Ptr&).
Usr _sqRoot(expr&)	In FB^3, this routine calls the standard Sqr routine.
Usr _lockBlk(hndl&)	Locks the indicated relocatable block. Same as Call HLock(hndl&).
Usr _fileAddr(fileID)	Returns the address of the i/o parameter block associated with the open file specified by fileID.
Usr _allocRelBlk(size&)	Allocates a new relocatable block which is size& bytes long, and returns a handle to the block. Same as Fn NewHandle(size&).
Usr _disposeBlk(hndl&)	Releases the relocatable block indicated by hndl&. Same as Call DisposeHandle(hndl&).
Usr _unlockBlk(hndl&)	Unlocks the indicated relocatable block. Same as Call HUnlock(hndl&).
Usr _sine(expr%)	In FB^3, this routine calls the standard Sin routine.
Usr _cosine(expr%)	In FB^3, this routine calls the standard Cos routine.

Example:

One of the most useful `Usr` functions is `Usr _fileAddr`. You can use the `Local Fn` shown here to display information about an open file.

```

Local Fn DisplayFileInfo(fileID)
  pbPtr& = Usr _fileAddr(fileID)
  Long If pbPtr& <> _nil
    Print "Filename = "; PStr$(pbPtr&.ioNamePtr&)
    Print "volRefNum = "; pbPtr&.ioVRefNum%
    Print "Dir ID  = "; pbPtr&.ioDirID&
  End If
End Fn

```

See Also:

`Sqr`; `Sin`; `Cos`; “File Manager” chapter in *Inside Macintosh: Files*

Usr Abs**function**

✓ *Appearance***✓** *Standard***✓** *Console*

Syntax:

```
AbsValue& = Usr Abs (expr)
```

Description:

This function returns the absolute value of *expr*. The absolute value of a number is its distance from zero: the absolute value of -23 is 23; the absolute value of 16 is 16. `Usr Abs` is slower than the `Abs` function, but it is included for backward compatibility.

See Also:

`Abs`

Usr AppleScriptGetResult

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
dataHandle& = Usr AppleScriptGetResult
```

Revision:

June, 2003 (Release 8)

Description:

This routine returns a handle pointing to the data in memory returned by an AppleScript script executed with `Usr AppleScriptRun` or `Usr AppleScriptLoadAndRun`.

If there is no result or if the routine fails to retrieve the data, the value returned by `Usr AppleScriptGetResult` is `_nil`. If your AppleScript script contains errors then *dataHandle*& will contain the lengthy error message returned by AppleScript. You are responsible for the disposing of the handle returned (if any).

You must be aware that the routine empties the AppleScript internal buffer. You could possibly use the `Usr AppleScriptGetResult` function to retrieve the text source of your entire script right after storing it into the AppleScript buffer using the `Route` and `Print` statements, however you cannot expect to compile and execute the script afterwards, since the buffer would be empty.

Note:

In a general manner, you must call the `Usr AppleScriptGetResult` function after a script has been executed whether an error occurred or not and dispose of the handle returned if any and if necessary.

The AppleScript routines are made available to your program with the following `Include` statement:

```
Include "Subs AppleScript.Incl"
```

Example:

The following example will return the list of all the folders located at the root level of your hard drive:

```

Include "Subs AppleScript.Incl"

Dim As Str255 message$
Dim As Handle textH
Dim As Rect r

Window 1
Call SetRect( r, 0,0, Window(_width), Window(_height) )
Call InsetRect( r, 32, 32 )
Edit Field 1, "",@r

Route _toAppleScript
Print "the name of every folder of the startup disk"
Route _toScreen

err = Usr AppleScriptRun(message$)
H = Usr AppleScriptGetResult
Long If H
    TESetText([H],Fn GetHandleSize(H), TEHandle(1))
    DisposeHandle( H )
    Cls
Xelse
    Edit$( 1 ) = message$
End If

```

See Also:

Usr AppleScriptLoadAndRun; Usr AppleScriptRun

Usr AppleScriptLoadAndRun function

✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
errorFlag = Usr AppleScriptLoadAndRun(fSpec, resID%)
```

Revision:

June, 2003 (Release 8)

Description:

This routine loads and executes the AppleScript script specified by *resID%* and located in the file specified by *fSpec*.

If the routine fails to launch the script *errorFlag* is set to a non-zero value. If the script returns a result, you can get a handle pointing to the data using the `Usr AppleScriptGetResult` function. Note that the script might also return an error in the AppleScript buffer, therefore you should always empty the buffer after a script has been executed getting the result with `Usr AppleScriptGetResult` then disposing of the handle returned if any.

fSpec is an FSSpec record specifying an existing file. If you pass `_nil` for the *fSpec* parameter, the script will be retrieved from the resource fork of the running application.

Note:

Usually, *resID%* is set to 128 which is the default resource ID for Apple's compiled scripts. However, if you pass 0 for this parameter, the routine will assume an ID of 128, thus:

```
Usr AppleScriptLoadAndRun(0,0)
```

is functionnally equivalent to

```
Usr AppleScriptLoadAndRun(appSpec,128).
```

In order for the script to function properly, your program must have entered its main event loop before it attempts to run the script.

The AppleScript routines are made available to your program with the following `Include` statement:

```
Include "Subs AppleScript.Incl"
```

See Also:

```
Usr AppleScriptGetResult; Usr AppleScriptStore
```

Usr AppleScriptRun

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
errorFlag = Usr AppleScriptRun(errorMessage$)
```

Revision:

June, 2003 (Release 8)

Description:

This function compiles and executes an AppleScript script that has been previously stored in FB's AppleScript internal buffer with the `Route` statement. The result returned by the routine is in the form of a flag that indicates whether the script was successfully compiled (`_noErr`) or if the operation has failed (non-zero value). The `errorMessage$` parameter will contain the result of the script (if any) or a human friendly error message from AppleScript.

The `errorMessage$` variable is set to contain up to 255 characters, consequently if the result of the script is larger than the maximum allowed, you will not get the full data returned by the script. In such case you may wish to use the `Usr AppleScriptGetResult` routine to get a handle to the entire result. In any case you must call that function to empty the AppleScript buffer and dispose of the possible handle returned.

Example:

The following example will return the file access path of the running application:

```
Include "Subs AppleScript.Incl"
Dim err
Dim reply$ // or Dim reply As Str255

Route _toAppleScript
Print "path to me As string"
Route _toScreen
err = Usr AppleScriptRun( reply$ )
If err = _noErr Then Print reply$
resultHandle = Fn AppleScriptGetResult
If resultHandle Then Call DisposeHandle( resultHandle )
```

Program output:

```
Macintosh HD:AppleScriptRun folder:FB_Temp
```

Note:

In order for the script to function properly, your program must have entered its main event loop before it attempts to run the script.

See Also:

```
Usr AppleScriptGetResult; Usr AppleScriptStore
```


Usr AppleScriptStore

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
errFlag = Usr AppleScriptStore(fSpec, resID%, scriptName$)
```

Revision:

August, 2002 (Release 7)

Description:

This routine saves a compiled script into the resource fork of the file specified by *fSpec* with a resource ID and a resource name specified respectively by *resID%* and *scriptName\$*. Once you have filled the AppleScript internal buffer with the `Route` and `Print` statements, you may wish to store the script permanently in compiled format for later use. The resulting script file can be run with the `Usr AppleScriptLoadAndRun` routine as well as with any application that can run AppleScript scripts such as Apple Script Editor.

fSpec is an `FSSpec` record that specifies a file. If the file doesn't exist, it is created. If you pass `_nil` for the *fSpec* parameter, the resulting compiled script will be stored in the resource fork of the running application.

resID% is usually set to 128 (the default resource ID for Apple's compiled scripts), and if you pass zero for this parameter the routine will assume an ID of 128. The routine will override any existing script having the same ID in the file specified. For example, `Usr AppleScriptStore(0,0,"My Script")` would store or replace in the resource fork of the running application a script with resource ID 128 whose resource name would be "My Script". You can pass a null string for the *scriptName\$* parameter.

Note:

In order for the script to function properly, your program must have entered its main event loop before it attempts to run the script.

The AppleScript routines are made available to your program with the following `Include` statement:

```
Include "Subs AppleScript.Incl"
```

See Also:

```
Usr AppleScriptGetResult; Usr AppleScriptLoadAndRun
```

Usr ConvertImageFile

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
err = Usr ConvertImageFile(srcName$, srcRef%,  
                           destName$, destRef%, newTypeIn&, newCreator&)
```

Revision:

June, 2001 (Release 5)

Description:

This function takes the picture information from one disk file and translates it into a different format. A new file is built with the correct file type and creator in the location that you determine.

If the final four parameters are left blank, QuickTime will bring up a "Save As..." dialog so that you may specify the type of file, its name, and its location.

Example:

The following example opens a graphic file and translates it into any specified format.

```
Include "Subs Image Files.Incl"  
  
gFBUseNavServices = _zTrue  
Dim fName$,err  
Dim @vRef%  
  
fName$ = Files$(_fOpen,, "Locate a picture",vRef%)  
  
Long If fName$[0]  
    err = Usr ConvertImageFile(fName$,vRef%,"",0,0,0)  
End If
```

Note:

Before you can use this function, you must include it in your project with the following statement:

```
Include "Subs Image Files.Incl"
```

See Also:

```
Usr ImageFileToPICT, Usr SaveImageFileAsPICT
```

Usr CopyFile

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
err = Usr CopyFile(SrcName$, SrcVol%, SrcDir&, -
                    DestName$, DstVol%, DstDir&)
```

Revision:

February, 2002 (Release 6)

Description:

This function copies a file from one location to another. If the copy is successful, the function returns `_noErr` (zero). If the copy fails, a file error code is returned. In the process of this operation, the original file is left intact. The source and destination folders may be the same if the file name is different.

Note:

You will not be able to use this function unless you include the proper header file as follows:

```
Include "Util_Files.incl"
```

Example:

The following example allows the user to select a file and a new destination before calling the `Usr CopyFile` function.

```
Include "Util_Files.incl"

Dim filename$, folderName$, destName$
Dim @vRefNum%, destVref%
Dim err

fileName$ = Files$(_fOpen, "Select a file to move", vRefNum)

Long If Len(fileName$)
    folderName$ = Files$(_fFolder, destVref%)
    Long If Len(folderName$)
        destName$ = "Copy of" + filename$
        err = Usr CopyFile(fileName$, vRefNum, -
                            0, destName$, destVref, 0)

        Long If err
            Print "File was not copied."
        Xelse
            Print "File copy successful."
        End If
    End If
End If
```

See also:

`Usr MoveFile`; `Rename`; `Files$`; `Usr CopyFile`

Usr Even

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
evenNumber& = Usr Even(expr)
```

Description:

This function returns the smallest even integer which is greater than or equal to the integer part of *expr*. `Usr Even` can be useful for finding even address boundaries.

Example:

```
Print Usr Even(5)  
Print Usr Even(-9)  
Print Usr Even(40.3)  
Print Usr Even(160)
```

Program output:

```
6  
-8  
40  
160
```

See Also:

`Mod`

Usr FileExists

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Boolean = Usr FileExists(fileName$, vRefNum%, dirID&)
```

Revision:

February, 2002 (Release 6)

Description:

Your program may determine whether or not a file exists using this function. A non-zero value is returned when the file is present. You need to pass the *fileName\$* and the *vRefNum%* in all cases, but the *dirID&* can usually be zero.

To determine if a file named "My File" is present in the application folder your would use the following code.

```
Long If Usr FileExists("My File", System(_aplVol), 0)
  Print "File exists!"
End If
```

Note:

You will not be able to use this function unless you include the proper header file as follows:

```
Include "Util_Files.incl"
```

See Also:

Usr FSFileExists

Usr FontHeight

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
fontHeight = Usr FontHeight
```

Description:

This function returns the current “font height” in pixels. This is based on the current font ID and font size for the current output window or printer.

The font height is the full height of a line of text, measured from the top of one line to the top of the line below it. It is not the same thing as the “font size,” although there is often a relationship between the font size and the font height. The font height is the sum of the font’s “ascent,” its “descent” and its “leading.” See the “Font Manager” chapter in *Inside Macintosh: Text* for more information.

Example:

`Usr FontHeight` is useful for determining how many lines of text will fit within a rectangle of a given height. This `Local Fn` returns the number of text lines that can fit in the current window:

```
Local Fn MaxLines
  wh = Window(_height)
  fh = Usr FontHeight
  max = wh \\ fh
End Fn = max
```

See Also:`Text`

Usr FSFileExists

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Boolean = Usr FSFileExists(fsSpec)
```

Revision:

February, 2002 (Release 6)

Description:

Your program may determine whether or not a file exists using this function. A non-zero value is returned when the file is present. You need to pass a file spec record which can be created using `Fn FBmakeFSSpec` or by using a file spec version of the `Files$` function.

To determine if a file named "My File" is present in the application folder your would use the following code:

```
Include "Util_Files.incl"

Dim fs As FSSpec
Fn FBmakeFSSpec(System(_aplVol),0,"My File",fs)
Long If Usr FSFileExists(fs)
    Print "File exists!"
End If
```

Note:

You will not be able to use this function unless you include the proper header file as follows:

```
Include "Util_Files.incl"
```

See Also:

Usr FileExists; Files\$; Appendix H: *File Spec Records*

Usr FSSendFileToTrash

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
error = Usr FSSendFileToTrash(FSSpec)
```

Revision:

February, 2002 (Release 6)

Description:

Rather than deleting a file, it is often desirable to move it to the trash. In this way, your user has a chance to retrieve it from the trash if he changes his mind. This particular version of the function requires a file spec record for operation. See Appendix H for information on file spec records.

Note:

You will not be able to use this function unless you include the proper header file as follows:

```
Include "Util_Files.incl"
```

See Also:

System function; Usr MoveFile; Rename; Appendix H: *File Spec Records*

Usr FSGetFolderName

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
resultCode = Usr FSGetFolderName(fsSpec, folderName$)
```

Revision:

August, 2002 (Release 7)

Description:

This function returns the name of the folder that contains the file specified by *fsSpec*.

Note:

You will not be able to use this function unless you include the proper header file as follows:

```
Include "Util_Files.incl"
```

Example:

```
Include "Util_Files.incl"
Dim fs As FSSpec
Dim x, name$

name$ = Files$(_FSSpecOpen,,,fs)

x = Usr FSGetFolderName(fs,name$)
Print name$
Do
    HandleEvents
Until 0
```

See Also:

FinderInfo, Files\$; Usr ScanFolder

Usr FSGetFullPathName

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
resultCode = Usr FSGetFullPathName(fsSpec, pathName$)
```

Revision:

August, 2002 (Release 7)

Description:

This function returns the full path name for the file specified by *fsSpec*. Folder names are separated by colons.

Note:

You will not be able to use this function unless you include the proper header file as follows:

```
Include "Util_Files.incl"
```

Example:

```
Include "Util_Files.incl"
Dim fs As FSSpec
Dim x,name$

name$ = Files$(_FSSpecOpen,,,fs)

x = Usr FSGetFullPathName(fs,name$)
Print name$
Do
    HandleEvents
Until 0
```

See Also:

FinderInfo, Files\$; Usr ScanFolder

Usr GetDoubleByte

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Boolean = Usr GetDoubleByte
```

Revision:

July 26, 2000 (Release 3)

Description:

If you have selected "Automate handlers for 1 & 2 byte strings" in the Preferences window under the Vars tab, this function will return a non-zero value when operating in double byte mode. Single byte mode is used for Roman fonts. Two byte script systems such as Chinese, Japanese, Vietnamese, and Korean require two bytes per character.

This function will return zero if you selected "Automate handlers for 1 & 2 byte strings" and if you have not used `Def SetSingleByte` (or have reset the system with `Def SetDoubleByte`.) In all other cases, it returns -1.

See Also:

```
Def SetDoubleByte; Def SetSingleByte, Usr GetSingleByte
```

Usr GetFullPathName

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
resultCode = Usr GetFullPathName(vRef%, dirID&, pathName$)
```

Revision:

August, 2002 (Release 7)

Description:

This function returns the full path name for a specified file. folder names are separated by colons.

Note:

You will not be able to use this function unless you include the proper header file as follows:

```
Include "Util_Files.incl"
```

Example:

```
Include "Util_Files.incl"
Dim @name$,vol%,dir
Dim err

name$ = Files$(_fOpen,,,vol%)

err = Usr GetFullPathName(vol%,0,name$)

Print name$

Do
  HandleEvents
Until 0
```

See Also:

Usr GetFolderName; FinderInfo, Files\$; Usr ScanFolder

Usr GetSingleByte

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Boolean = Usr GetSingleByte
```

Revision:

July 26, 2000 (Release 3)

Description:

If you have selected "Automate handlers for 1 & 2 byte strings" in the Preferences window under the Vars tab, this function will return a non-zero value when operating in single byte mode. Single byte mode is used for Roman fonts. Two byte script systems such as Chinese, Japanese, Vietnamese, and Korean require two bytes per character.

This function will return zero if you selected "Automate handlers for 1 & 2 byte strings" and if you have not used `Def SetSingleByte`. In all other cases, it returns -1.

See Also:

```
Def SetDoubleByte; Usr GetDoubleByte, Def SetSingleByte
```

Usr GetPICT**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
pictHandle& = Usr GetPICT(({rect|#rectAddr&})
```

Description:

This function captures a picture of a portion of the screen using the screen's current color depth, and returns a handle to the picture. Given the handle, you can then draw the picture (using the `Picture` statement), or save the picture as a "PICT" resource (using the `AddResource` Toolbox routine), etc.

The captured picture is bound by the rectangle which is specified in *rect* (which should be an 8-byte variable such as a `Rect` type), or which is pointed to by *rectAddr*& (which should be a long-integer expression or a `Pointer` variable). The rectangle should be expressed in the local coordinate system of the current output window.

The captured picture is also limited to the "clip region" of the current output window. Therefore, what actually gets captured is the intersection of the clip region and the specified rectangle.

The captured picture is not limited to the visible contents of the window. Whatever is on the screen within the specified rectangle (and within the clip region) will get captured. By appropriately adjusting the rectangle and the clip region, you can capture any part of the screen you want.

Example:

This program captures a snapshot of the entire screen, then displays the snapshot at 1/2 scale in a small window.

```

Dim rect.8, pt.4
sWidth  = System(_scrnWidth)
sHeight = System(_scrnHeight)
Window -1,"", (0,0)-(sWidth/2, sHeight/2), _dialogMovable
HandleEvents      'Let Finder refresh things
'Get screen corner in window's local coord's:
Call SetPt(pt, 0, 0)
Call GlobalToLocal(pt)
'Define rect as entire screen, in local coord's:
Call SetRect(rect, pt.h%, pt.v%, pt.h%+sWidth, pt.v%+sHeight)
'Save window's current clip region:
oldClip& = Fn NewRgn
Call GetClip(oldClip&)
'Set clip region to entire screen:
Call ClipRect(rect)
'Snap entire screen:
h& = Usr GetPICT(rect)
'Restore old clip region:
Call SetClip(oldClip&)
Call DisposeRgn(oldClip&)
'Draw the little picture:
Window 1          'make window visible
HandleEvents      'Handle initial refresh event
Picture (0,0)-(sWidth/2, sHeight/2), h&
Def DisposeH(h&)

Do
    HandleEvents
Until _false

```

This Local Fn takes a snapshot of the current window's contents, then pastes the picture onto the clipboard. Note that if the window is hidden or partially obscured, other parts of the screen will appear in the captured picture.

```

Local Fn Window2Clip
    Dim rect.8
    Call SetRect(rect, 0, 0, Window(_width), Window(_height))
    oldClip& = Fn NewRgn          'Init a new region
    Call GetClip(oldClip&)        'Save a copy of current clip region
    Call ClipRect(rect)           'Set clip region to this rectangle
    pictH& = Usr GetPICT(rect)    'Get the picture
    Call SetClip(oldClip&)        'Restore the old clip region
    Call DisposeRgn(oldClip&)     'Don't need this copy any more
    Long If Fn ZeroScrap = _noErr 'Clear clipboard contents
        Call HLock(pictH&)
        OSErr = ▯
        Fn PutScrap(Fn GetHandleSize(pictH&), _"PICT", [pictH&])
    End If
    Kill Picture pictH&           'Don't need this any more
End Fn

```

See Also:

Picture

Usr Handle2Btn

function✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
butTonID% = Usr Handle2Btn(ctrlRecHandle&)
```

Description:

If *ctrlRecHandle&* is the handle to the Control Record of a FutureBASIC-created button or scrollbar, `Usr Handle2Btn` returns the ID number of the button or scrollbar (this is the number that your program assigned to the button or scrollbar in the `Button` statement or `Scroll Button` statement). `Usr Handle2Btn` returns the ID number regardless of whether the control is in the current output window or in a different window; if the control is in the current output window, then `Usr Handle2Btn` is the inverse of the `Button&` function.

If *ctrlRecHandle&* is not a valid handle to a Control Record, or it's the handle to a control that was created by other means than a `Button` statement or a `Scroll Button` statement, then the value returned by `Usr Handle2Btn` is undefined.

See Also:

`Button&`; `Usr WPtr2WNum`

Usr ImageFileToPICT

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
pictHandle% = Usr ImageFileToPICT(fileName$, vRef%)
```

Revision:

June, 2001 (Release 5)

Description:

Use this simple function to read the contents of a graphic image from the disk and convert it into a picture handle. When you are finished with the handle, you will need to dispose of it using Kill Picture.

```
Include "Subs Image Files.Incl"

gFBUseNavServices = _zTrue
Dim fName$,err
Dim @vRef%
Dim pHndl As Handle

fName$ = Files$(_fOpen,,"Locate a picture",vRef%)

Long If fName$[0]
  Window 1
  pHndl = Usr ImageFileToPICT(fName$,vRef%)
  Long If pHndl
    Picture (0,0), pHndl
    Kill Picture pHndl
  Xelse
    Print "Could not convert image file."
  End If
End If

Do
  HandleEvents
Until 0
```

Note:

Before you can use this function, you must include it in your project with the following statement:

```
Include "Subs Image Files.Incl"
```

See Also:

```
Usr ConvertImageFile, Usr SaveImageFileAsPICT
```

Usr MoveFile

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
err = Usr MoveFile(SrcName$, SrcVol%, SrcDir&, DstVol%, DstDir&)
```

Revision:

February, 2002 (Release 6)

Description:

This function moves a file from one location to another. If the move is successful, the copy of the file at the original location is erased and the function returns `_noErr` (zero). If the move fails, a file error code is returned.

Note:

You will not be able to use this function unless you include the proper header file as follows:

```
Include "Util_Files.incl"
```

Example:

The following example allows the user to select a file and a new destination before calling the `Usr MoveFile` function.

```
Include "Util_Files.incl"

Dim filename$, folderName$
Dim @vRefNum%, destVref%
Dim err

fileName$ = Files$(_fOpen,, "Select a file To move", vRefNum)

Long If Len(fileName$)
    folderName$ = Files$(_fFolder,,, destVref%)

    Long If Len(folderName$)
        err = Usr MoveFile(fileName$, vRefNum, 0, destVref, 0)
        Long If err
            Print "File was not moved."
        Xelse
            Print "File move successful."
        End If
    End If
End If
```

See also:

`Usr MoveFile`; `Rename`; `Files$`; `Usr CopyFile`

U_{sr} OpenRFP_{erm}**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
resRef = Usr OpenRFPerm(fileName$, vRef%, permission)
```

Revision:

August, 2002 (Release 7)

Description:

With the release of Mac OS X, the ability to easily open a resource file has dissipated like smoke. But a simple modification will revive your older `Fn OpenRFPerm`. Just change the `Fn` to `Usr` and a new FB routine will work in version of the System software from System 7.x to Mac OS X.

As with the original toolbox function, the parameters are the file name, the volume reference number (which is handled in the background by FB) and the permission constant. Legal permission constants are:

```
_fsRdPerm
_fsRdWrPerm
_fsRdWrShPerm
_fsCurPerm
_fsRdPerm
_fsWrPerm
```

Usr ReplaceResource

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
rHndl& = Usr ReplaceResource(rHndl&, rTp&, rID%, rName$, rRef%)
```

Revision:

August, 2002 (Release 7)

Description:

This is probably one of the most useful functions in all of Resourcedom. It will add or replace a resource in a file specified by the *rRef* parameter. If the original handle is a resource (normally a sure ticket to MacsBug) it will duplicate the old handle to side step potential problems. If the resource exists, the data in the handle is substituted for the existing resource. If the resource does not exist, it is added.

rHndl& The handle that will become a resource

rTp& A 4 character OSType such as `_"PICT"` or `_"TEXT"`

rID% The new resource ID number

rName\$ The name for the new resource. If this is a null string and the resource exists, the old name will survive the change.

rRef% The resource file's reference number. If you want the resource added to the current application, use `System(_aplRes)`

See Also:

```
Def ChangedResource; Usr OpenRFPerm
```

Usr Round, RoundUp, RoundDown

function

✓ Appearance

✓ Standard

✓ Console

Syntax:

```
wholeNumber = Usr Round(numericExpression)
wholeNumber = Usr RoundUp(numericExpression)
wholeNumber = Usr RoundDown(numericExpression)
```

Description:

- RoundRounds the specified *numericExpression* to a whole number using the rounding rule (where .5 and higher are rounded up and lower values are rounded down).
- RoundUpRounds the specified *numericExpression* to the next whole number.
- RoundDownRounds the specified *numericExpression* down to the nearest whole number. This is the same as `Fix`

Example:

```
Dim i#
For i = 1 To 2 Step 0.1
  Print i#, Usr RoundDown(i#), Usr Round(i#), Usr RoundUp(i#)
Next
```

Program output:

1	1	1	1
1.1	1	1	2
1.2	1	1	2
1.3	1	1	2
1.4	1	1	2
1.5	1	2	2
1.6	1	2	2
1.7	1	2	2
1.8	1	2	2
1.9	1	2	2

See Also:

Abs; Fix; Frac; Int

Usr SaveImageFileAsPICT

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
err% = Usr SaveImageFileAsPICT(srcName$, srcVol%, destName$, destVol%)
```

Revision:

June, 2001 (Release 5)

Description:

Use this simple function to convert a graphic image file into a PICT format. *destName\$* indicates the name of the converted file and *destVol%* its location. The following example allows to select a graphic file whose format is recognized by QuickTime and save the converted file with a PICT format.

```
Include "Subs Image Files.Incl"

Dim fName$,vRef%,err

fName$ = Files$(_fOpenPreview,"qtif",,vRef%)
Long If fName$[0]
    err = Usr SaveImageFileAsPICT(fName$,vRef%,
                                fName$ + " Converted",vRef%)
    Print "You now have a file named:"fName$" Converted"
    Stop
End If
```

Note:

Before you can use this function, you must include it in your project with the following statement:

```
Include "Subs Image Files.Incl"
```

See Also:

```
Usr ImageFileToPICT, Usr ConvertImageFile
```

Usr ScanFolder

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
resultCode = Usr ScanFolder(FileScanRecord)
```

Revision:

February, 2002 (Release 6)

Description:

In order to use this function, you must first create a file scan record. The format for this record can be found in the "Util_Files.incl" header and is as follows:

```

Begin Record FileScanRec
  Dim ScanSpec As FSSpec ' 70 bytes
  Dim ScanIndex As Long ' File Index
  Dim Type As OSType ' File Type
  Dim CreaTor As OSType ' Creator Type
  Dim DataLen As Long ' Data Fork Size
  Dim RsrcLen As Long ' Resource Fork Size
  Dim CDate As Long ' Creation Date
  Dim MDate As Long ' Modification Date
  Dim BDate As Long ' Backup Date
  Dim FndrLoc As Long ' Finder Location
  Dim FndrFlags As word ' File Flags
  Dim FileAttr As word ' File Attributes
  Dim recursive As word ' True if recursive scan
End Record

```

You dimension such a record like this:

```
Dim myFileScanRecord As FileScanRec
```

To begin a scan, you will need to set up a couple of values in the scan record. The first is the FSSpec portion of the record, the second is the index which starts at zero for the first file in a folder.

```

Dim fsr As FileScanRec
fsr.ScanIndex = 0
fsr.ScanSpec = myFSSpecRec

```

Each time this statement is executed, the ScanIndex value is incremented and information from the next file in the list is extracted. It is not necessary to manually handle this value, but it is possible to do so when desired.

Detecting Folders:

If the scanning process detects a folder, the result code of 1 is returned and the file type is set to `_fldr`. You may use subsequent calls of `Usr ScanFolder` to examine embedded folders. If you make these recursive calls, be sure to set the recursive field of the `FileScanRec` to `_zTrue`.

Example:

The following example lets the user select a file using `Files$`. Then all of the files in the same folder (along with some of the information contained in the `FileScanRec`) are displayed.

```

Include "Util_Files.incl"

Dim fsr As FileScanRec
Dim fs As FSSpec
Dim fileName$

fileName$ = Files$(_FSSpecOpen,""," ",fs)

fsr.ScanSpec = fs
fsr.ScanIndex = 0

while Usr ScanFolder(fsr) >= 0
  Long If fsr.FndrFlags And _finvisible
    Print fsr.ScanIndex,"Not Visible"
  Xelse
    Print fsr.ScanIndex,mki$(fsr.Type),
    Print fsr.DataLen,"";fsr.ScanSpec.name;""
  End If
wend

```

See Also:

`Files$`; `Rename`; `Usr MoveFile`; `Usr CopyFile`

Usr SendFileToTrash

function

*✓ Appearance**✓ Standard**✓ Console*

Syntax:

```
error = Usr SendFileToTrash(fileName$, vRefNum%)
```

Revision:

February, 2002 (Release 6)

Description:

Rather than deleting a file, it is often desirable to move it to the trash. In this way, your user has a chance to retrieve it from the trash if he changes his mind.

Note:

You will not be able to use this function unless you include the proper header file as follows:

```
Include "Util_Files.incl"
```

See Also:

System **function**; Usr MoveFile; Rename, Usr FSSendFileToTrash

Usr StrOffset

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
offset& = Usr StrOffset(element%,resID%|Handle&)
```

Revision:

September 17, 2000 (Release 4)

Description:

Returns the offset position (in bytes) to the beginning of a string element specified by *element%* in a STR# resource with an ID of *resID%* or from a handle that matches the STR# resource style.

Example:

```
Dim t$,offset&,h&
h& = Fn GetResource("STR#",-20481)
Long If h&
    offset& = Usr StrOffset(65,-20481)
    Print PStr$([h&]+offset&)
End If
```

Program output:

```
Could not locate the AppleGuide help file. To access the help file,
you need to have it installed in the Extensions folder inside your
System Folder.
```

See Also:

Str#, StringList

Usr WPtr2WNum**function**✓ *Appearance*✓ *Standard*✗ *Console***Syntax:**

```
WindowID% = Usr WPtr2WNum(WindowPtr&)
```

Description:

If *WindowPtr&* is the pointer to the window record of a FutureBASIC-created window, `Usr WPtr2WNum` returns the window's ID number (this is the number that your program assigned to the window in the `Window` statement). `Usr WPtr2WNum` is the inverse of the `Get Window` statement and the `Window(_wndPointer)` function.

If *WindowPtr&* does not point to a window record, or it points to a window record of a window that was created by other means than the `Window` statement, the `Usr WPtr2WNum` function returns zero.

See Also:

`Get Window`; `Window` function; `Handle2Btn`

Val

function

✓ Appearance

✓ Standard

✓ Console

Syntax:

`numericValue = Val(string$)`

Description:

If *string\$* contains the characters of a number in any of the standard FB^3 formats (decimal, hex, octal or binary), Val returns the number's value.

Val ignores leading spaces in *string\$*. When it finds a non-space character, it evaluates the remaining characters in *string\$* until it encounters a character which is not part of the number. Thus, for example, the string " 3245.6" would be evaluated as 3245.6, but the string " 32 W45.6" would be evaluated as 32. If the first non-space character in *string\$* can't be recognized as part of a number, Val returns zero. Val performs the opposite of functions such as Str\$, Hex\$, Oct\$, Bin\$ and Uns\$.

Example:

```
Data "-3.2", "1.4E2", "&4C1", "9+7"
For i = 1 To 4
  Read s$
  Print s$, Val(s$)
Next
```

Program output:

-3.2	-3.2
1.4E2	140
&4C1	1271
9+7	9

Note:

If *string\$* represents an integer, consider using the Val& function, which is faster.

See Also:

Val&; Mki\$; Cvi; Str\$; Hex\$; Oct\$; Bin\$; Uns\$; Appendix C: Data Types and Data Representation

Val&**function**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
integerValue& = Val&(string$)
```

Description:

This function is similar to the `Val` function, but it can only evaluate a string which represents an integer. The advantage of using `Val&` is that it executes more quickly than `Val`.

`string$` can represent an integer in decimal, hex, octal or binary format. The absolute value of the represented integer must not exceed 4,294,967,295.

`Val&` ignores leading spaces in `string$`, and it stops evaluating the string when it encounters a character that is not part of standard integer notation. Note that this means `Val&` will stop evaluating the string when it encounters a decimal point or an “E” exponent indicator. That means that certain strings which represent integers will be evaluated differently by `Val` than by `Val&`. For example, the string “24.61E2” will be evaluated as 2461 by `Val`, but as 24 by `Val&`.

See Also:

`Val`; `Mki$`; `Cvi`; `Str$`; `Hex$`; `Oct$`; `Bin$`; `Uns$`; [Appendix C: Data Types and Data Representation](#)

ValidRect

function✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
ignored = Fn ValidRect(rect)
```

Revision:

February, 2002 (Release 6)

Description:

Before Carbon became a part of the Mac toolbox, we were able to use a toolbox procedure called `ValidRect` to mark a portion of the current window as an area that did not need to be refreshed during the next update. This call will not work in OS-X (or in the Carbon version of OS 9). Our substitute, `Fn ValidRect`, will work in versions 7 through X without additional coding required on your part.

VarPtr

function

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
address& = VarPtr ({var|Fn userFunction})
address& = @var
```

Description:

`VarPtr(var)` returns the memory address where the first byte of the variable `var` is located. You can use this value as a “pointer” to `var`. If `var` is a local variable inside a local function, the value returned by `VarPtr(var)` may be different each time you execute the function, and is not valid after the function exits. The syntax `@var` is just a shorthand version of `VarPtr(var)`.

`VarPtr(Fn userFunction)` is identical to the `@Fn userFunction` function.

Note:

You cannot use `VarPtr(var)` with variables that use register storage, because such variables do not have addresses. See the `Dim` statement and the `Register On/Off` statements to learn how to prevent a variable from using register storage.

Because the “@” symbol has a special meaning when it appears after the `Print` or `LPrint` keyword, you cannot use the `@var` syntax as the first item in a list of print items.

```
Print @myVar#           'This does not work ("@" is misinterpreted)
Print (@myVar#)         'This works.
Print VarPtr(myVar#)    'So does this.
```

See Also:

@Fn; Register On/Off; Dim; Print; LPrint; Peek; Poke; BlockMove

Wend**statement**

See the `While` statement.

While

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```

While expr
    [statementBlock]
Wend

```

Description:

The `While` statement marks the beginning of a “while-loop,” which must end with a `Wend` statement. *statementBlock* represents a block of zero or more executable statements, possibly including other while-loops. When a `While` statement is encountered, FB^3 evaluates *expr*. If *expr* is nonzero, FB^3 executes the statements in *statementBlock*; otherwise it jumps down to the first statement following `Wend`.

If the *statementBlock* statements are executed, the process is repeated; *expr* is evaluated again, and if it’s still nonzero, the *statementBlock* statements are executed again. This loop continues until *expr* becomes zero, at which point the program exits the loop and jumps down to the first statement following `Wend`.

Typically, *expr* is an expression involving logical operators, which is evaluated either as `_zTrue` (−1) or as `_false` (0). See the `If` statement for more information about *expr*.

Note that if *expr* is zero the first time it’s evaluated, the statements in *statementBlock* are not executed at all. If you want to use a looping structure which will always execute *statementBlock* at least once, consider using a `Do...Until` loop or a `For...Next` loop.

See Also:

`For...Next`; `Do...Until`; `If`

Width**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Width [LPrint] [=] {_noTextWrap|_textWrap|numChars}
```

Description:

This statement affects how (and whether) text printed by subsequent `Print` or `LPrint` statements will “wrap.” If you specify the `LPrint` keyword, the `Width` statement applies only to statements sent to the printer. If you omit the `LPrint` keyword, the `Width` statement applies only to subsequent `Print` statements destined for the screen. `Width` (without `LPrint`) applies to all existing and subsequently-created windows.

While “wrapping” is enabled, any subsequently printed text whose location exceeds a certain limit on the current line will automatically “wrap around” and continue at the beginning of the next line. Wrapping does not necessarily occur on word boundaries.

If you specify `_noTextWrap`, wrapping is disabled. Text continues on the current line until the pen is explicitly moved to the next line (this usually happens automatically after the last item in the `Print` or `LPrint` statement has been printed). Note that if the window or the printer page is not wide enough to display all of the items in the print list, some of the items will be lost. The advantage of using `_noTextWrap` is that it greatly increases printing speed.

If you specify `_textWrap`, wrapping occurs at the right edge of the window or the printer page. This is the default condition in effect before the first execution of `Width`.

If you specify `numChars` (which must be a number in the range 1 through 255), wrapping occurs either at the right edge of the window (or the printer page), or after `numChars` characters have been printed on the current line, whichever occurs first. Note that if you’re using a proportional font, the horizontal pixel location where wrapping occurs may be different on different lines.

Console behavior:

When you use the Console runtime, the `Width` statement is ignored; essentially, `Width [LPrint] _textWrap` is always in effect. Also, previously-printed text in the Text Window will dynamically “re-wrap” as the user resizes the window. If you’re not using the Console runtime, you won’t get the dynamic re-wrap effect; previously-printed text stays where it is, and may be obliterated if the user resizes the window.

See Also:

`LPrint`; `Print`; `Route`

Window

function

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
WindowInformation = Window(expr)
```

Revision:

February, 2002 (Release 6)

Description:

This function returns information related to a window (usually the current output window). The value you specify in *expr* determines what kind of information is returned, as described in the following paragraphs.

ID's of Active Window , Active Document Window, Active Palette Window and Output Window

- `Window(_activeWnd)` returns the window ID number of the currently active window, or zero if no window is active.
- `Window(_activeDoc)` returns the window ID number of the currently active document window or zero if no document window is active. In searching for the active document, this function bypasses all palettes in search of a window with the type attribute set to include `_keepInBack`.
- `Window(_activePlt)` returns the window ID number of the frontmost palette. In order for there to be a palette, one or more document windows must be open with the type attribute set to include `_keepInBack`. At that point, all non-`_keepInBack` windows become palettes and float over the document windows.
- `Window(_outputWnd)` returns the window ID number of the current output window, or zero if output is currently directed to somewhere besides a FutureBASIC-created screen window (e.g., to the printer).

Window Size

- `Window(_width)` returns the width (in pixels) of the content region of the current output window.
- `Window(_height)` returns the height (in pixels) of the content region of the current output window.

(Note: The content region does not include the window's frame.)

Window Position (Appearance manager)

- `Window(_kFBstructureTop)` returns the distance from the top of the screen to the top of the structure region of the window.
- `Window(_kFBstructureLeft)` returns the distance from the left of the screen to the left of the structure region of the window.
- `Window(_kFBstructureWidth)` returns the width of the window's structure region.
- `Window(_kFBstructureHeight)` returns the height of the window's structure region.
- `Window(_kFBcontentTop)` returns the distance from the top of the screen to the top of the content region of the window.
- `Window(_kFBcontentLeft)` returns the distance from the left of the screen to the left of the content region of the window.
- `Window(_kFBcontentWidth)` returns the width of the window's content region. This is normally the same as `Window(_width)`.
- `Window(_kFBcontentHeight)` returns the height of the window's content region. This is normally the same as `Window(_height)`.

Pen Position

- `Window(_penH)` returns the horizontal position (in pixels) of the pen in the current output window.
- `Window(_penV)` returns the vertical position (in pixels) of the pen in the current output window.

Window Record Pointer

- `Window(_wndPointer)` or `Window(_wndRef)` returns a pointer to the Window Record of the current output window. For information about the contents of the Window Record, see the `Get Window` statement, and the “Window Manager” chapter of *Inside Macintosh: Macintosh Toolbox Essentials*, as well as the descriptions of `grafPort` and `CGrafPort` data structures in *Inside Macintosh: Imaging with QuickDraw*.
- `Window(_wndPort)` return the current `grafport` being used for output.

Clipboard Contents

- `Window(_textClip)` returns a nonzero value if there is information of type “TEXT” on the clipboard; returns zero otherwise.
- `Window(_pictClip)` returns a nonzero value if there is information of type “PICT” on the clipboard; returns zero otherwise.

Window Class (Standard BASIC only)

- `Window(_outputWClass)` returns the “class number” assigned to the current output window.
- `Window(_activeWClass)` returns the “class number” assigned to the currently active window.
- `Window(_outputWCategory)` returns the “class number” assigned to the current output window for the Appearance Manager runtime.
- `Window(_activeWCategory)` returns the “class number” assigned to the currently active window for the Appearance Manager runtime.

(See the `Window` statement for more information about class numbers).

Other Window Info (Appearance Manager)

- `Window(_kFBMacWClass)` returns the toolbox window class. Return values might include things like `_kDocumentWindowClass` or `_kMovableModalWindowClass`
- `Window(_kFBMacWAttributes)` returns toolbox attributes about a window. Values might include `_kWindowResizableAttribute` or `_kWindowCloseBoxAttribute`
- `Window(_kFBwDescHandle)` returns the handle to a `FBWindowDescription` record which is stored in the window's refcon. While this information is subject to change, it currently contains the following data:

```

Begin Record FBWindowDescription
    Dim FBwRef           As Long    // standard FB Ref number
    Dim FBwZoomRect      As Rect    // best zoom out pos,
                                     // empty for default

    Dim FBwAttributes    As WindowAttributes
    Dim FBwWindowClass   As WindowClass
    Dim FBwControlList    As Handle // linked list of controls
    Dim FBwEFList         As Handle // linked list of EFs
    Dim FBwFSSpec         As FSSpec // associated file spec
                                     // (affects proxy icon)

    Dim FBwCreaTor        As OSType // for proxy icon
    Dim FBwFileType       As OSType // for proxy icon
    Dim FBidealSizeX      As Short   // zoom
    Dim FBidealSizeY      As Short   // zoom
    Dim FBwClipRgn        As RgnHandle
    Dim FBwVScrollH       As Handle  // 0 If no v scroll bar
    Dim FBwHScrollH       As Handle  // 0 If no h scroll bar
    Dim FBwCategory       As Long
    Dim FBwClickThru      As Boolean
    Dim FBwUpdateVisRgn   As Boolean
    Dim FBwKeepInactive   As Boolean // for backdrop Window
    Dim FBwNoAuToFocus    As Boolean // affects tab key
                                     // handling with EFs &
                                     // text buttons

    Dim &

```

End Record

- `Window(_kFBwClickThru)` returns a non-zero value if this attribute bit is set.
- `Window(_kFBFloatingWndPtr)` returns the window pointer of the frontmost floating window.

Screen Borders in Local Coordinates

- `Window(_toLeft)` returns the horizontal pixel position of the screen's left edge, expressed in the local coordinate system of the current output window (note this will be negative if the window lies entirely on the screen).
- `Window(_toTop)` returns the vertical pixel position of the top of the screen, expressed in the local coordinate system of the current output window (note this will be negative if the window lies entirely on the screen).
- `Window(_toRight)` returns the horizontal pixel position of the screen's right edge, expressed in the local coordinate system of the current output window.
- `Window(_toBottom)` returns the vertical pixel position of the bottom of the screen, expressed in the local coordinate system of the current output window.

(Note that these numbers are meaningless if output is currently directed to some place other than a screen window.)

Checking Whether a Window Exists

If you specify a negative value in `expr`, `Window(expr)` returns a nonzero value if there exists a window whose ID number is `Abs(expr)`; it returns zero otherwise. The returned value does not depend on whether the window is currently visible or not; it only depends on whether the window has been created (using the `Window` statement) and not yet closed (using the `Window Close` statement).

Edit Field and Picture Field Information

- `Window(_efNum)` returns the ID number of the currently active edit field or picture field; or zero if there is no currently active edit field or picture field.
- `Window(_selStart)` returns the character position of the beginning of the selected text or insertion point in the currently active edit field (if any).
- `Window(_selEnd)` returns the character position of the end of the selected text or insertion point in the currently active edit field (if any).
- `Window(_efHandle)` returns a handle to the `TextEdit` record of the currently active edit field (if any); this is the same as the value returned by `TEHandle(Window(_efNum))`.
- `Window(_lastEfNum)` returns the ID number of the previously active edit field (or zero, if no other edit field was previously active).
- `Window(_efTextLen)` returns the number of characters in the currently active edit field (if any).
- `Window(_teBlock)` returns a handle to the Edit Field Descriptor for the currently active edit field or picture field (if any).
- `Window(_efClass)` returns the `efClass` parameter assigned to the currently active edit field (if any); or the *negative* of the *just* parameter assigned to the currently active picture field (if any).

Note:

If output is currently directed to a graphics port other than a screen window (e.g. to the printer, or to an offscreen `GWorld`), then references to the “current output window” apply to the current port, unless otherwise specified.

See Also:

`Window` statement; `Edit Field`; `Picture Field`; `SetSelect`; `TEHandle`;
`Get Window`; `System function`; `Appearance Window`; `Def WindowCategory`

Window

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
Window [#]idExpr[, [title$][, [rect][, [type][, [class]]]]
```

Description:

Use this statement to do any of the following:

- Create a new screen window;
- Activate (highlight and bring to the front) an existing window;
- Make an existing window visible or invisible;
- Alter the title or rectangle of an existing window.

The parameters should be specified as follows. They are interpreted slightly differently depending on whether you are creating a new window or altering an existing one.

- *idExpr* a positive or negative integer whose absolute value is in the range 1 through 255.
- *title*\$ a string expression.
- *rect* a rectangle in global screen coordinates. You can express it in either of two forms:
 - $(x1, y1) - (x2, y2)$ Two diagonally opposite corner points.
 - *rectAddr*& Long integer expression or `Pointer` variable which points to an 8-byte struct such as a `Rect` type.
- *type* a positive or negative integer which specifies the general appearance of the window, and specifies whether it should be “modal” or not (a modal window is always active while it’s open; it inhibits the user from selecting another window or a menubar item until the window is closed).
- *class* an integer in the range 0 through 255.

To Create a New Screen Window

- Specify an *idExpr* value such that `Abs(idExpr)` is different from the ID number of any existing window. A new window is created and is assigned an ID number of `Abs(idExpr)`. You can use the window's ID number later to identify the window in other FB^3 statements and functions. If *idExpr* is negative, the window is created invisibly; it's sometimes useful to create a window invisibly if it will contain controls, edit fields and graphics that may take a long time to build. You can use the `Window` statement again to make an invisible window visible (see below). When you create a new window, it becomes the current output window. If you create it visibly (and you don't specify the `_keepInBack` attribute), it also becomes the current active window.
- *title\$* assigns a string to the window's title bar (if the window has a title bar). If you omit this parameter, the window will be created without a title.
- *rect* specifies the initial size and location of the window's content rectangle. Note that *rect* does not include the window's frame. This parameter is interpreted in a special way if you specify an upper-left coordinate of (0,0) in *rect*; in this case, the window is centered in the screen, and its width and height are determined by the right and bottom coordinates of *rect*. Note that this special interpretation applies only when you're creating a new window. If you omit this parameter, a window of a "default" size and location is created.
- *type* specifies the appearance, modality and special attributes of the window (see more below). If you omit this parameter, a non-modal window of type `_doc` is created, with no special attributes.
- *class* specifies an optional "class number" for the window. If your application creates several windows, it's useful to assign the same class number to each window that performs a given kind of function. Later, you can use the `Window` function to determine the class number of the currently active window and the current output window; this can help your application determine what the window is used for and how to process it. If you omit this parameter, the new window is assigned a class number of zero.

To Activate an Existing Window

- Specify the (positive) ID number of an existing window in *idExpr*. You do not need to specify any of the other parameters, unless you also wish to change some of the window's characteristics. The window also becomes the current output window. If the window was invisible, it becomes visible.

Note: You can't activate the window if you specified the `_keepInBack` attribute when the window was created, and there are other visible windows open.

To Make an Existing Window Visible or Invisible

- To make a window visible, specify the (positive) ID number of an existing window in *idExpr*. The window also becomes the current active window (unless its `_keepInBack` attribute is set), and it becomes the current output window.
- To make a window invisible, specify the negative of an existing window's ID number in *idExpr*. The window becomes the current output window. If it was the active window, it becomes inactive (possibly forcing another window to become active).

You do not need to specify any of the other parameters, unless you also want to change some of the window's characteristics.

To Alter the Characteristics of an Existing Window

- Specify the ID number of an existing window (or its negative) in *idExpr*, and specify a new *title\$* and/or *rect* parameter (you can't change the window's *type* nor *class* after it's been created). If you omit any parameter, the corresponding characteristic won't change. Note that the *rect* parameter is interpreted slightly differently when you're altering an existing window, as opposed to creating a new window; in particular, specifying an upper-left coordinate of (0,0) will *not* cause an existing window to be centered on the screen. If you want to change an existing window's rectangle so that it's centered on the screen, use a *rect* parameter that's calculated as follows:

```
Dim rect.8
x1 = (System(_scrnWidth) - myWindowWidth) / 2
y1 = (System(_scrnHeight) - myWindowHeight) / 2
x2 = x1 + myWindowWidth
y2 = y1 + myWindowHeight
Call SetRect(rect, x1, y1, x2, y2)
```

Note: If you specify the window's (positive) ID number when you alter a window's characteristics, the window also becomes the current active window (unless its `_keepInBack` attribute is set). If you specify the negative of the window's ID number, the window becomes invisible.

Side Effects of Activating the Window

The `Window` statement always makes the window active, unless you specify a negative *idExpr*, or you specified the `_keepInBack` attribute when you created the window. When you activate a window using the `Window` statement, the following things also happen:

- The window also becomes the current output window. (See the `Window Output` statement to learn how to specify an output window that's different from the active window.)
- A `Dialog` event of type `_wndActivate` is generated. (There are also other kinds of actions which generate `_wndActivate` events; see the `Dialog` function for more information.)
- Any previously-active window becomes inactive (this also generates a separate `_wndActivate` `Dialog` event).

Side Effects of Making a Window Visible

If *idExpr* is the (positive) ID of a window that currently exists but is invisible, the `Window` statement makes the window visible, and also generates a `Dialog` event of type `_wndRefresh`. A `_wndRefresh` event is also generated when you create a new window visibly. (Note: There are also other kinds of actions which generate `_wndRefresh` events; see the `Dialog` function for more information.)

Side Effects of Making a Window Invisible

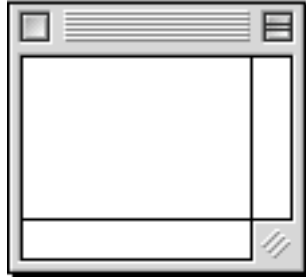
If *idExpr* is negative, the window becomes invisible, and it becomes the current output window. If the window was previously active, it becomes inactive; if your program has other visible windows, one of them becomes the active window.

More about the type Parameter

The *type* parameter is a positive or negative integer. The absolute value of *type* determines the general appearance of the window and determines some special attributes of it. The sign of *type* determines whether the window will be modal or non-modal. If *type* is negative, the window will be modal, which means the user won't be able to switch to a different window until the modal window is closed; if the user clicks in an inactive window while a modal window is active, a beep is generated, but a `_wndClick` event is not generated. *type* can be expressed as follows:

```
[ - ] ( WindowVariant [ +attribute [ +attribute ... ] ] )
```

WindowVariant can be any of the following:



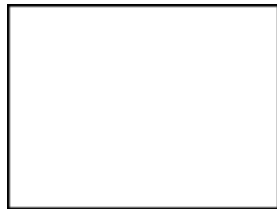
`_doc` (1)

Document window with room for scrollbars.



`_dialogFrame` (2)

Framed dialog window.



`_dialogPlain` (3)

Plain dialog window.



`_dialogShadow` (4)

Shadowed dialog window.

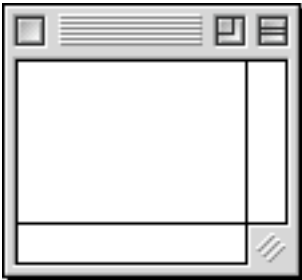


`_docNoGrow` (5)

Document window with no grow box.



`_dialogMovable` (6) Movable dialog window.



`_docZoom` (9) Document window with zoom box and room for scrollbars.



`_docRound` (17) Rounded-corner window. Add 1 through 7 to this variant to increase the roundness of the corners.



`_WDEFbaseID` (129) Palette window with close box. You can add the constant `_WDEFhasZoom` (8) to this variant to give the window a zoom box.



`_WDEFbaseID + _WDEFsideDrag` (131) Sideways palette. You can add the constant `_WDEFhasZoom` (8) to this variant to give the window a zoom box.

176 through 191 These values are mapped to Appearance Manager window types 1984 through 1999.

192 through 255 These values are mapped to Appearance Manager window types 1024 through 1087.

The *attribute* can be any of the following (you can specify as many of these as apply):

<i>attribute</i>	<i>Value</i>	<i>Description</i>
<code>_noGoAway</code>	256	Creates a window without a “go-away” box (a close box).
<code>_keepCtrlsActive</code>	512	Keeps buttons, edit fields, and other controls highlighted even when the window is not active. If you don’t specify this attribute, the controls etc. will dim when the window becomes inactive (but see also the <code>_keepInBack</code> and <code>_keepWndActive</code> attributes).
<code>_noAutoClip</code>	1024	If you specify this attribute, the window’s edit fields, picture fields and controls can be obliterated and/or drawn over; if you don’t specify this attribute, such regions are protected. See also the <code>AutoClip</code> statement and the <code>Window Fill</code> statement.
<code>_updateVisRgn</code>	2048	This attribute affects how the window’s clip region will be set when FutureBASIC3 calls your dialog-event handling routine with a <code>_wndRefresh</code> event. If you specify this attribute, the clip region will be set to include only that part of the window which was identified as actually needing a refresh (the clip region will be reset to its previous value when the routine exits). If you omit this attribute, the clip region will be set to include the entire window (possibly excluding controls, edit fields, etc.)
<code>_clickThru</code>	4096	This attribute affects what happens when your program activates the window in response to a <code>_wndClick</code> event. If the <code>_clickThru</code> attribute is set, the activating click will be “passed through” to the window; this may cause other events (such as <code>_btnClick</code> or <code>_efClick</code>) to be generated, depending on what was clicked on. If you omit this attribute, two separate clicks are required to activate the window and to interact with its contents.
<code>_keepInBack</code>	8192	Setting this attribute has the following effects: <ul style="list-style-type: none"> • If there are other visible windows open, this window cannot be made active, and cannot be brought to the front. • Clicking in the window never generates a <code>_wndClick</code> event, but causes the window to become the current output window the next time <code>HandleEvents</code> is executed. • The window, and its controls, edit fields, etc. are always highlighted. • Clicks in the window are “passed through” to the controls, etc., generating events as if the window were active. <p>This attribute is useful for windows which need to remain behind a smaller “floating” window such as a palette window.</p>
<code>_keepWndActive</code>	16384	If you specify this attribute, the window and its controls remain highlighted even when the window is inactive. This is similar to the <code>_keepCtrlsActive</code> attribute, but it also keeps the window’s frame highlighted as well as its controls.

See Also:

`MinWindow`; `MaxWindow`; `SetZoom`; `Get Window`; `Window Close`; `Window Output`;
`Window function`; `Dialog function`; `AutoClip`

Window Close

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Window Close [#]WindowID
```

Description:

This statement closes the window whose ID number is *WindowID*, removing it from the screen. It also closes all edit fields, picture fields, buttons and other controls that were in the window. If you re-use the same *WindowID* value in a subsequent *Window* statement, a new window is created.

If you're closing the active window, and your program has other visible windows open, one of the other windows becomes the active window, and becomes the current output window. If you're closing the current output window (but it's not the active window), you should explicitly designate a new destination for output (using the *Window* statement or the *Window Output* statement) before executing any subsequent text or drawing commands.

See Also:

Window statement

Window Fill

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
Window Fill [[#]expr]
```

Description:

This statement immediately redraws, in the current output window, any controls, edit fields and picture fields which intersect with the window's "invalid region." The window's invalid region consists of those visible parts of the window which were recently uncovered but haven't yet been refreshed; you can also add an arbitrary rectangle to the invalid region using the new (in Release 6) `Fn InvalRect`.

After redrawing the controls etc., `Window Fill` resets the window's invalid region to a nil (empty) region. This will inhibit FB^3 from generating a `_wndRefreshDialog` event for the window the next time `HandleEvents` is executed.

If no window number is used for `expr`, FB^3 assumes that you wish to use the current output window.

Note:

You will rarely need to use this statement. FB^3 automatically redraws controls, edit fields and picture fields as necessary every time your program executes `HandleEvents`.

See Also:

`AutoClip`; `Dialog` function

Window Output

statement

✓ <i>Appearance</i>	✓ <i>Standard</i>	✗ <i>Console</i>
---------------------	-------------------	------------------

Syntax:

```
Window Output [#]idExpr
```

Description:

This statement makes an existing window the current output window. The current output window is the destination for any subsequent text and drawing commands, and is the target window for such statements and functions as `Button`, `Edit Field`, etc.

The *idExpr* can be either a positive or negative number; `Window Output` affects the window whose ID number is `Abs(idExpr)`. If you specify a negative *idExpr*, the window becomes invisible. If you specify a positive *idExpr*, the window becomes visible.

`Window Output` does not activate the specified window (i.e., it does not highlight its contents nor bring it to the front), if there are other visible windows open. Use the `Window` statement when you want to explicitly activate a window. (Note: the `Window` statement also makes the specified window the current output window.)

`Window Output` is useful in cases where you need to update the contents of a window which may be in the background, without bringing the window to the front.

See Also:

`Window statement`; `Window function`

Window Picture

statement

✓ *Appearance*✓ *Standard*✗ *Console*

Syntax:

```
Window Picture [#]WindowID, pictHandle&
```

Description:

This statement “attaches” the picture whose handle is *pictHandle&* to the window specified by *WindowID*. The picture is refreshed automatically as necessary, and all `_wndRefresh` Dialog events for the window are inhibited. This is sometimes useful when you want to display a window which shows nothing but a single unchanging picture. However, because `_wndRefresh` events are inhibited, it makes it difficult to maintain other kinds of contents in the window. It’s generally more useful to use the `Picture Field` statement instead.

Note:

You must not dispose or release the picture in *pictHandle&* until you first either close the window (using `Window Close`), or detach the picture from the window. You can detach the picture either by explicitly attaching a different picture, or by executing the following:

```
Window Picture [#]WindowID, _nil
```

See Also:

`Picture statement`; `Picture function`; `Picture On/Off`; `Picture Field`; `Dialog function`

WndBlk**function***✗ Appearance**✓ Standard**✗ Console***Syntax:**

```
Ptr& = WndBlk
```

Description:

This function returns a pointer to FutureBASIC's internal Window Descriptor Block. The Window Descriptor Block is an array of Window Descriptors. Each Window Descriptor is a record which contains information about one of your program's windows; FutureBASIC3 maintains a Window Descriptor for each window that you create using the `Window` statement. The structure of a Window Descriptor is as follows:

```
Begin Record wDescriptor
  Dim wptr&           'Pointer to window record
  Dim efdHeadH&       'Handle to the first edit field descriptor
  Dim activeEfH&      'Handle to active edit field
  Dim clipH&          'Handle to window's clip region
End Record
```

- The `wptr&` element points to the window's window record, which is a structure that the MacOS Window Manager maintains for each open window. This is the same pointer that is returned by the `Get Window` statement.
- The `efdHeadH&` element is a handle to the first Edit Field Descriptor for the window. There is an Edit Field Descriptor for each edit field or picture field in the window; see below for more information. If there are no edit fields nor picture fields in the window, `efdHeadH&` is `_nil` (zero).
- The `activeEfH&` element is a handle to the currently active edit field or picture field (if any) in the window. This is the same as the handle returned by the `Window(_efHandle)` function when the window is active. If there is no active field in the window, `activeEfH&` is `_nil` (zero).
- The `clipH&` element is a handle to the window's clip region. However, if the window's `_noAutoClip` attribute is set, `clipH&` is `_nil` (zero).

You can use the window's assigned ID number to determine the location of its Window Descriptor within the array. The first Window Descriptor in the array corresponds to "window #0," which is a special graphics port that is reserved for FutureBASIC's internal use. The next element in the array corresponds to window #1; the next to window #2; and so on. Array elements that don't correspond to the ID of any existing window have undefined contents.

If you define a `wDescriptor` record type as shown above, you can get a pointer to the Window Descriptor corresponding to window `#WindowID` as follows:

```
wdPtr& = WndBlk + (WindowID * SizeOf(wDescriptor))
```

Alternatively, you can use `Xref` to access the contents of the array using normal FutureBASIC3 array syntax:

```
wdArray& = WndBlk
Xref wdArray As wDescriptor
```

You can then use the syntax `wdArray(WindowID)` to access an individual Window Descriptor.

Edit Field Descriptors

FutureBASIC3 maintains an Edit Field Descriptor for each field that you create using the `Edit Field` statement or the `Picture Field` statement. For each window, there is a linked list of Edit Field Descriptors for the fields in that window; a handle to the first descriptor in the list can be found in the `efdHeadH&` element of the Window Descriptor record. The structure of an Edit Field Descriptor is as follows:

```
Begin Record efDescriptor
  Dim nextDescH&      'Handle to next descriptor in list
  Dim indexRef%       'Field's ID number
  Dim fieldType``     'Field's type
  Dim jClass``        'Text justification & class number
  Dim teH&            'Handle to TextEdit record
End Record
```

- The `nextDescH&` element is a handle to the next Edit Field Descriptor in the list. If `nextDescH&` is `_nil`, there are no more edit fields nor picture fields in this window.
- The `indexRef%` element is the field's ID number, as assigned by the `Edit Field` statement or the `Picture Field` statement.
- The `fieldType``` element is the same as the type parameter specified in the `Edit Field` statement or the `Picture Field` statement (it defaults to `_framedNoCR (1)` if no type was explicitly specified).
- The `jClass``` element is the same as the `efClass` parameter specified in the `Edit Field` statement, or the `just` parameter specified in the `Picture Field` statement. It defaults to zero if the parameter wasn't specified.
- The `teH&` element is a handle to the field's TextEdit Record, which is a structure that the MacOS maintains for each open edit field. This is the same handle that is returned by the `TEHandle(efID)` function. Note that FutureBASIC3 associates a TextEdit Record with picture fields as well as edit fields; you can use a picture field's TextEdit Record, for example, to determine the field's rectangle. See the "TextEdit" chapter of Inside Macintosh: *Text* for more information.

Example:

The following function returns a count of the number of edit fields and picture fields in a specified window. It assumes that the `wDescriptor` and `efDescriptor` record types have been defined as shown above.

```

Local Fn EfCount(wndID)
  Dim wdPtr As Pointer To wDescriptor
  Dim efdH As Handle To efDescriptor
  count = 0
  If Window(-wndID) = 0 Then Exit Fn ' (no such window)
  wdPtr = WndBlk + (wndID * SizeOf(wDescriptor))
  efdH = wdPtr.efdHeadH&
  While efdH <> _nil
    Inc(count)
    efdH = efdH..nextDescH&
  Wend
End Fn = count

```

Note:

Window Descriptors are maintained only for windows that are created using the `Window` statement, and Edit Field Descriptors are maintained only for edit fields/picture fields that are created using the `Edit Field` or `Picture Field` statement. If you create a window or an edit field using a MacOS Toolbox routine (such as `NewWindow` or `TENew`), there will be no Descriptor associated with it.

See Also:

`Window statement`; `Edit Field`; `Picture Field`; `Get Window`; `TEHandle`

Write# statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Write[#] deviceID,{var|stringVar$;len} [, {var|stringVar$;len}...]
```

Description:

This statement writes the contents of the specified variables to the open file or serial port specified by *deviceID*, starting at the current “file mark” position. The variables in the list can be of any type (including record types). If you specify a string variable, it must be followed by a *len* parameter, which indicates the number of bytes to copy from the string. *len* can be any positive numeric expression whose value doesn’t exceed the length of the string.

Unlike the `Print#` statement, the `Write#` statement does not apply any formatting to the data before writing it. Instead, it simply writes an exact copy of the variables’ internal bit patterns to the device. This makes the written data suitable for input by the `Read#` statement. In general, the data written by `Write#` is not suitable for files which are to be interpreted as text.

See Also:

`Print#`; `Read#`; `Input#`; `Open`

Write Dynamic

statement✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Write Dynamic deviceId, arrayName
```

Revision:

May, 2001 (Release 5)

Description:

Use `Write Dynamic` to send the contents of a dynamic array to a disk file. Data written to a file in this manner can be read back into memory using `Read Dynamic`. Before dynamic arrays are written to the disk, they are automatically compressed.

Example:

```

Dim x
Dynamic myAry(_maxLong)

For x = 1 To 100
    myAry(x) = x
Next

Open "O",#1,"Dynamic Array Test"
Write Dynamic #1,myAry
Close #1

Kill Dynamic myary

Open "I",#1,"Dynamic Array Test"
Read Dynamic #1,myAry
Close #1

For x = 1 To 10
    Print myary(x)
Next

Kill "Dynamic Array Test"
```

See Also:

Compress Dynamic, Dynamic, Read Dynamic, Write Dynamic

Write Field

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Write Field[#] deviceID, Handle&
```

Description:

Use this statement to write the contents of the relocatable block specified by *Handle*& to the open file or serial port specified by *deviceID*, in a format suitable for input by the `Read Field` statement.

`Write Field` starts writing at the current “file mark” position. It first writes a 4-byte long-integer which indicates the size of the block; following this, it writes the contents of the block itself.

Note:

If you want to write only the block’s contents to the file (without the 4-byte length indicator), use the `Write File` statement instead:

```
Write File# deviceID, [Handle&], Fn GetHandleSize(Handle&)
```

See Also:

`Read Field`; `Get Field`; `Write File`; `Open`

Write File

statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Write File[#] deviceId, address&, numBytes&
```

Description:

This statement writes *numBytes*& of data to the open file or serial port specified by *deviceId*, starting at the current “file mark” position. The data is copied from the memory which starts at *address*&. This is generally the fastest way to write a large amount of data to a file.

Example:

This program fragment saves the binary image of an array into an open file. You can later use the `Read File` statement to quickly load the array using the data in the file (see the example accompanying the `Read File` statement).

```
_maxSubscript = 200
Dim myArray%(_maxSubscript)
arrayBytes& = (_maxSubscript+1) * SizeOf(Int)
:
Write File #1, @myArray%(0), arrayBytes&
```

See Also:

`Open`; `Write#`; `Write Field`; `Read File`

Xelse**statement**

See the Long If s and End If tatements.

Xor

operator

✓ *Appearance*

✓ *Standard*

✓ *Console*

Syntax:

```
result& = exprA {Xor | ^^} exprB
```

Description:

Expression *exprA* and expression *exprB* are each interpreted as 32-bit integer quantities. The `XOR` operator performs a “bitwise comparison” of each bit in *exprA* with the bit in the corresponding position in *exprB*. The result is another 32-bit quantity; each bit in the result is determined as follows:

<i>Bit value in exprA</i>	<i>Bit value in exprB</i>	<i>Bit value in resultat&</i>
0	0	0
1	0	1
0	1	1
1	1	1

A common use for `XOR` is to toggle the state of individual bits in a bit pattern. For example:

```
pattern& = pattern& Xor Bit(7)
```

This flips bit 7 in `pattern&` from 0 to 1 or from 1 to 0, and leaves all of `pattern&`'s other bits alone.

Example:

The example below shows how bits are manipulated with `xor`:

```
DefStr Long
Print Bin$(923)
Print Bin$(123)
Print "-----"
Print Bin$(923 Xor 123)
```

Program output:

```
000000000000000000000000000000001110011011  
000000000000000000000000000000001111011  
-----  
000000000000000000000000000000001111100000
```

See Also:

And; Or; Not

Xref statement

✓ *Appearance*✓ *Standard*✓ *Console*

Syntax:

```
Xref arrayName(maxSub1[,maxSub2 ...]) [As dataType]
```

Description:

The `Xref` statement declares an array, and associates the array with the memory pointed to by a particular long-integer variable, called the “link variable.” You can use `Xref` to cause any arbitrary block of memory to be treated as an array. This is especially useful when you need to dynamically create an array whose size can’t be determined until runtime, or when you want to impose an array structure on data that was created outside of your FB^3 program.

The link variable must be a simple (non-array, non-field) long-integer variable which has the same name as the array (ignoring any type-identifier suffix). For example, if you specify the following:

```
Xref diameter#(3,7)
```

The compiler creates a long-integer variable called `diameter&`. When you run the program, you should set `diameter&` equal to some appropriate memory address (you do this after the `Xref` statement); FB^3 then assumes that the `diameter#()` array begins at that address. When you examine elements in the array, they are retrieved from the memory pointed to by `diameter&`. When you alter elements in the array, the memory pointed to by `diameter&` is altered.

The first subscript is arbitrary

The `maxSub1`, `maxSub2` etc. values must be positive static integer expressions. However, since `Xref` does not actually allocate any memory, the declared subscripts are used somewhat differently than in a `Dim` statement. The second and subsequent subscripts (if any) determine the internal structure of the array, and they should exactly match the internal layout of the elements pointed to by the link variable. But the value of the first subscript (`maxSub1`) is basically ignored, and may be arbitrarily set to any value greater than zero. When you actually reference the array elements, you can use subscript values that are larger than `maxSub1`, as long as they reference valid elements within the block of memory pointed to by the link variable.

Note:

`Xref` is a non-executable statement, so you can't change its effect by putting it inside a conditional-execution structure such as `Long If...End If`. However, you can conditionally include it or exclude it from the program by putting it inside a `Compile Long If...Compile End If` block. The `Xref` statement should appear somewhere above the first line where the array is referenced.

Warning:

XREFs will not behave properly when used with handles.

```
Xref myHandleList(100) As Handle : Rem will not work!
```

See Also:

```
Dim; Xref@
```

Xref@**statement**✓ *Appearance*✓ *Standard*✓ *Console***Syntax:**

```
Xref@ arrayName(maxSub1[,maxSub2 ...]) [As dataType]
```

Description:

Xref@ is identical to the **Xref** statement, except that the link variable is interpreted as a handle, rather than as a pointer. You should use **Xref@** when you want the contents of a relocatable block to be treated as an array.

Example:

The following declares an array called `inclination!`, allocates a new block with enough room for `numElts&` elements, and associates the `inclination!` array with the contents of the block.

```
Xref@ inclination!(1)
inclination& = Fn NewHandle(numElts& * SizeOf(Single))
```

Note that, because the value of `maxSub1` is ignored in the **Xref@** statement, we can arbitrarily set it to 1. However, when we actually reference the elements of the `inclination!` array, we can specify any subscript value in the range 0 through `numElts& - 1`.

See Also:

`Dim; Xref`



Appendices

Appendix A: Specifying Files and Directories

There are a number of FB³ statements and functions in which you must specify a particular file and/or a particular directory. A directory is basically a folder; in addition, every volume (basically, every disk) has a root directory, which has the same name as the volume itself. A directory is a container; when you double-click on a folder icon in the Finder, you see a window which shows that directory's contents. Likewise, when you double-click on a volume's icon, you see a window which shows the contents of that volume's root directory.

As the MacOS has evolved, a number of different schemes have emerged for specifying files and directories. The statements and functions in FB³ use a system which involves three different pieces of information, which interact in ways that are sometimes subtle. These pieces of information are: the Path Name, the Directory Reference Number, and the Directory ID Number.

The Path Name

In the MacOS, a path name is a string of no more than 255 characters, consisting of colons and/or names of files and directories in a particular format. It's usually identified as *path\$* in syntax descriptions in the manual.

- A *full path name* begins with the name of a volume, and ends with the name of a file or directory that is somewhere within that volume. In between is a hierarchical list of the directories leading from the root level to the final item; the names in the list are separated by colons. If the last item in the list is a directory, it may optionally be followed by a final colon. A full path name *always* contains at least one colon, but never begins with one. A full path name which consists of just the volume's name followed by a colon, indicates the volume's root directory.
- A *partial path name* either begins with a colon, or consists of just a single item's name with no colons. It indicates a hierarchical directory path *relative to* some "base directory" (discussed below). A partial path name that contains no colons indicates an item at the base directory's first level. A partial path name that consists *only* of a colon (with no item names listed) refers to the base directory itself. A partial path name may also begin with *multiple* colons: a leading double-colon indicates that the path begins at the base directory's *parent*; a leading triple-colon indicates that the path begins at the base directory's "grandparent," and so forth.

The Directory Reference Number

This is a peculiar concept which, for historical reasons, can be interpreted in a number of different ways depending on its value. It can also be interpreted as indicating either a volume, or a particular directory on a volume, depending on the value of another quantity called the "Directory ID Number" (discussed below). The Directory Reference Number is always stored as a "signed short-integer" number, so it can range in value from -32768 to +32767. It's usually identified as *refNum%* in syntax descriptions in the manual.

<i>refNum%</i>	<i>Directory interpretation</i>	<i>Volume interpretation</i>
Zero (or not specified)	The current default directory	The volume containing the current default directory
Positive. <i>refNum%</i> is interpreted as a Drive ID number of a disk drive.	The root directory of the volume (if any) in that disk drive	The volume (if any) in that disk drive
–1 through –16383. <i>refNum%</i> is interpreted as a Volume Reference Number.	The root directory on the indicated volume	The indicated volume
–16384 through –32768. <i>refNum%</i> is interpreted as a Working Directory Reference Number.	The indicated Working Directory	The volume containing the indicated Working Directory

(Note: A Drive ID Number is assigned to each disk drive when the system starts up. A Volume Reference Number is assigned to each volume when it's mounted, and is valid until the volume is unmounted. A Working Directory Reference Number is assigned to a particular directory on a particular mounted volume at the request of your application, and is valid until your application "closes" that directory (as with the `Close Folder` statement), or your application quits.)

The Directory ID Number

This is a positive number that the MacOS assigns permanently to each directory at the time the directory is created. On any given volume, no two Directory ID Numbers are the same; however, two directories on two different volumes might both have the same Directory ID Number.

In those FB^3 statements that use a Directory ID Number (`Open`, `Kill`, `Name`, `Rename`), you can use two different techniques to specify the number:

- You can specify it explicitly as a parameter to the statement. It's usually identified as *dirID&* in syntax descriptions in the manual.
- If you don't specify it explicitly, FB^3 uses the number that you've specified in the `ParentID` statement. If you haven't yet executed the `ParentID` statement, FB^3 uses zero. Also, the `Open`, `Kill`, `Name` and `Rename` statements always automatically reset the `ParentID` value back to zero after the statement executes.

Note: certain other FB^3 statements and functions (such as `Folder`) do not use a Directory ID Number and don't pay attention to the `ParentID` value. Another way to put it is: such statements and functions always behave as though the Directory ID Number were zero.

How These Three Things Interact

FB³ uses the Path Name, the Directory Reference Number and the Directory ID Number in the following ways to identify which file or directory you're indicating:

- If the Path Name is a *full path name*, then the Directory Reference Number and Directory ID Number are ignored. The item is completely specified by the Path Name. (Note that if there are two mounted volumes which both have the *same volume name*, then a full path name can be ambiguous.)
- If the Path Name is a *partial path name* (or is not specified), then the Directory Reference Number and Directory ID Number are used together to identify a *base directory*. The item, as identified by the partial path name, is then located relative to that base directory (or, if no Path Name is specified, the item is the base directory itself). The base directory is determined as follows:
 - If the Directory ID Number is *zero*, then the base directory is determined entirely by the Directory Reference Number, according to the "Directory Interpretation" column in the table above.
 - If the Directory ID Number is *nonzero*, then both the Directory Reference Number and the Directory ID Number are used to identify the base directory. In this case, the base directory's *volume* is indicated by the Directory Reference Number (according to the "Volume Interpretation" column in the table), and the Directory ID Number identifies a directory within that volume.

Other Ways to Specify Items

When you specify files or folders in calls to MacOS Toolbox routines, you don't always need to use the three data items discussed above. The preferred way to identify a file or folder in many Toolbox calls is by means of a fileSpec, which is a 70-byte record containing a volume reference number, a directory ID number, and the item's name. To track an item which may not be on a currently mounted volume, or which may have been renamed or moved to a different folder, the preferred method is to use an alias record. Refer to *Inside Macintosh: Files* for a complete discussion of fileSpec's and alias records.

Note that many MacOS Toolbox routines also support the use of Path Names, Directory Reference Numbers (usually called `ioVRefNum`), and Directory ID Numbers, using exactly the same rules of interaction discussed above.

Appendix B:

Variables

In FB^3, a variable can be thought of as a named container for data. The “container” has a specific size and (usually) a specific address in memory. Also, each variable has a specific type which determines how FB^3 interprets its contents (See Appendix C). You can copy data into a variable by putting the variable on the left side of the “=” symbol in a `Let` statement; or by explicitly modifying the contents at the variable’s address (through statements like `Poke` and `BlockMove`). Certain other FB^3 statements and functions (such as `Swap` and `Inc`) may also modify a variable when you include the variable as a parameter. In FB^3, a variable can have any of the following forms:

- `identifier[tiSuffix]`

A simple string or numeric variable, such as: `myLong&`, or `theString$.tiSuffix` is the optional type-identifier suffix, such as “\$”, “%”, “&”, etc. See the `Dim` statement, and Appendix C: *Data Types and Data Representation*, for a complete list of type-identifier suffixes. Examples:

```
myIntVar
xyz&`
```

- `stringVar$(offset)` (Note: the square brackets are part of the variable)

This variable consists of the single byte which is located at `offset` bytes past the beginning of the string variable `stringVar$`. (The \$ is required) This variable’s type is `Unsigned Byte`. This kind of variable is normally used to quickly retrieve or alter a single character in a string. The statement, “`x = stringVar$(offset)`” is equivalent to: “`x = Peek(@stringVar$ + offset)`”. The statement, “`stringVar$(offset) = x`” is equivalent to: “`Poke @stringVar$ + offset, x`”. Examples:

```
firstname$(3)
```

- `pointerVar`

A pointer variable. This is an identifier declared as a `Pointer` type; it can be declared either as a “generic” pointer, or a pointer to some other specific type. Examples:

```
myPtr
anotherPtr
```

- `handleVar`

A handle variable. This is an identifier declared as a `Handle` type; it can be declared either as a “generic” handle, or a handle to some other specific type. Examples:

```
myHandle
thisHdl
```

- `recordName`

The variable is an entire record. This can be either a “pseudo-record” (declared using `Dim recordName.constant`), or a “true record” (declared using `Dim recordName AS recordType`). Examples:

```
myTrueRec
iopb
```

- `arrayName[tiSuffix] (expr1 [,expr2...])`

The variable is a specific element of an array. This can be an array of any type, but `tiSuffix` can only be used in numeric or string arrays. Note that an entire array is not considered to be a variable. Examples:

```
firstName$(15)
recArray(3, x%)
```

- `pseudoRecordName.const1[.const2...]tiSuffix`

This variable consists of the bytes located at a specific offset from the beginning of a “pseudo-record.” `const1`, `const2` etc. are previously-defined non-negative symbolic constant names (minus their leading underscore character), or non-negative integer literals. The address of this variable is at $(const1 + const2 + \dots)$ bytes past the beginning of the pseudo-record. The size and type of this variable are determined by `tiSuffix` (for example, if `tiSuffix` is “&”, then the variable is a 4-byte signed long integer). Examples:

```
House.streetName$
pBlock.ioDrUsrWds.frRect.left%
```

- `psRecArray.const1[.const2...]tiSuffix(expr1 [,expr2...])`

This variable consists of the bytes located at a specific offset from the beginning of a specific element in an array of “pseudo-records.” The address of this variable is at $(const1 + const2 + \dots)$ bytes past the beginning of the array element. The size and type of this variable are determined by `tiSuffix`. Examples:

```
HouseArray.streetName$(42,6)
pb.rect.bottom%(z)
```

- `addressVar&.const1[.const2...]tiSuffix`

The variable consists of the bytes located at a specific offset from the address given in `addressVar&`. The address of this variable is at $(const1 + const2 + \dots)$ bytes past the given address; the size and type of this variable are determined by `tiSuffix`. `addressVar&` must be a (signed or unsigned) long-integer variable, or a generic `Pointer` variable. `addressVar&` must be a “simple” variable; it cannot be an array element nor a record field. Examples:

```
recPtr&.myField%`
genericPtr.rectangle.right%
```

- `handleVar&..const1[.const2...]tiSuffix`

The variable consists of the bytes located at a specific offset from the beginning of the relocatable block referenced by `handleVar&`. The address of this variable is at $(const1 + const2 + \dots)$ bytes past the beginning of the block. The size and type of this variable are determined by `tiSuffix`. `handleVar&` must be a (signed or unsigned) long-integer variable, or a generic `Handle` variable. `handleVar&` must be a “simple” variable; it cannot be an array element nor a record field. Examples:

```
recHdl&..thisField.thatField$
genericHandle..someField``
```

Variables involving fields of “true records”

The fields of a “true record” are defined inside a `Begin Record...End Record` block. A field’s declared data type can be any valid type; if a field is itself declared as another “true record” type, then the field can have “subfields,” which are just the fields of that secondary record.

A field can also be declared as an array (of any type). In this case, whenever the field’s name is included as part of a variable specification, it must be followed by subscript(s) in parentheses. Thus, in each of the variable descriptions listed below, each `field` and `subfield` takes one of the following forms, depending on whether or not it’s an array field:

For non-array fields:

```
field/subfield ::= fieldName[tiSuffix]
```

For array fields:

```
field/subfield ::= fieldName[tiSuffix] (sub1 [,sub2...])
```

The type and size of each of the following variables is just the type and size of the last `field` or `subfield` specified.

- `trueRecordName.field[.subfield ...]`

The variable is the specified field or subfield of the specified “true record.” Examples:

```
myTrueRec.myField%
stats.game(7).teamName$(1)
```

- `recordPtr.field[.subfield ...]`

The variable is the specified field or subfield of the “true record” pointed to by `recordPtr`. The `recordPtr` must be declared as a pointer to a specific type of record. Examples:

```
ptr1.myField
ptr2.arrayField$(3)
```

- `recordHdl..field[.subfield ...]`

The variable is the specified field or subfield of the “true record” referenced by `recordHdl`. The `recordHdl` must be declared as a handle to a specific type of record. Examples:

```
Hdl1..book(3).title$
Hdl2..phoneNum
```

- `arrayName(expr1[,expr2 ...]).field[.subfield ...]`

This variable is the specified field or subfield of a specific element in an array of “true records.” Examples:

```
HouseArray(42,6).streetName$
season(2).game(3).player(6)
```

- `ptrArray(expr1[,expr2 ...]).field[.subfield ...]`

This variable is the specified field or subfield in a “true record” pointed to by an element in an array of pointers. The array must be declared as an array of pointers to a specific type of record. Examples:

```
myPtrArray(n).field3&
ptrArray(6,2).miscInfo.chapter(7).title$
```

- `handleArray(expr1[,expr2 ...])..field[.subfield ...]`

This variable is the specified field or subfield in a “true record” referenced by an element in an array of handles. The array must be declared as an array of handles to a specific type of record.

Examples:

```
myHndlArray(7,j)..map  
myHndlArray(7,j)..map.quadrant(x,3).icon&
```

Limitations

There are some limitations on how many variables can be assigned.

Arrays are limited to about 2 gigabytes (each).

Simple variables inside of a local function are limited to 32K (per local function).

The .MAIN file of a project often allocates variables outside of local functions that are not global. These are treated as variables for a local function and are limited to 32K.

Appendix C: Data Types and Data Representation

Revision:

May 30, 2000 (Release 3)

I. Integers

Integers can be represented as literals, as symbolic constants, or as variables.

I.1 Integer Literals

- *Decimal*: a string of decimal digits, optionally preceded by “+” or “-”.

Examples: 7244 -328442

- *Hexadecimal*: a string of up to 8 hexadecimal digits, preceded by “&” or “&H” or “0x” (that’s a zero-x). Hexadecimal digits include the digits 0 through 9, and the letters A through F. Letters can be either in upper or lower case.

Examples: &H12a7 0x47BeeF &42AD9

- *Octal*: a string of up to 10 octal digits, preceded by “&O” (that’s the letter “O”, not a zero). Octal digits include the digits 0 through 7.

Examples: &o70651 &o32277

- *Binary*: a string of up to 32 binary digits, preceded by “&x”. Binary digits include the digits 0 and 1.

Examples: &x0100011 &x10110000111011001

- *Quoted*: a string of up to 4 characters, surrounded by double-quotes, with an underscore preceding the initial quote. Each character in the quoted string represents 8 bits in the internal bit pattern of the resulting integer, according to the character’s ASCII code.

Examples: _"TEXT" _"N*"

Note: Hexadecimal, octal, binary and quoted literals reflect the actual bit patterns of the integers as they’re stored in memory. These may be interpreted either as positive or negative quantities, depending on which types of variables they’re assigned to. If they’re not assigned to any variable, they’re generally interpreted as positive quantities.

I.2 Symbolic Constants

A symbolic constant is an identifier preceded by an underscore character. There are many symbolic constants which have pre-defined values in FB^3. You can also define your own symbolic constants within your program, either by using a `Begin Enum...End Enum` block; or a `Dim Record...End Record` block; or by using a “constant declaration” statement. A constant declaration statement has this syntax:

```
_constantName = staticExpression
```

where `_constantName` is a symbolic constant which has not been previously defined, and `staticExpression` is a “static integer expression” (see Appendix D: *Numeric Expressions*). The value of `staticExpression` must be within the range -2,147,483,648 through +2,147,483,647. Once a symbolic constant has a value assigned to it, that value cannot be changed within your program. Like all constants, a symbolic constant has a global scope.

A constant declaration may also include pascal style strings using one of the following formats:

```
_constantName$ = "I am a string constant"
_constantTab$  = 9  : Rem Chr$(9) = tab character
_constantCR$   = 13 : Rem Chr$(13) = carriage return
_twoByteKanjiChar = 10231: Rem KChr$(10231)
```

I.3 Integer Variables

There are six different types of integer variables in FB^3; they differ in the amount of storage space they occupy, and in the range of values they can represent. An integer variable's name may end with a type-identifier suffix which indicates its type; alternatively, you can declare an integer variable's type by using the `As` clause in a `Dim` statement. If a variable has no type-identifier suffix, and wasn't declared with an `As` clause, then FB^3 checks whether there are any `Def<type>` statements which are applicable to the variable. Finally, if the variable can't be typed by any of the above means, FB^3 assigns the type "signed short integer" to the variable. Arrays of integers, and integer record fields, are typed by similar means.

<i>Type</i>	<i>Storage</i>	<i>Range</i>	<i>Type identification</i>
signed byte	1 byte	-128..+127	x' Dim x As Byte Dim x As Char
unsigned byte	1 byte	0..255	x` Dim x As Unsigned Byte Dim x As Unsigned Char
signed short integer	2 bytes	-32768..+32767	x% Dim x As Int Dim x As Word Dim x As Short
unsigned short integer	2 bytes	0..65535	x%` Dim x As Unsigned Int Dim x As Unsigned Word Dim x As Unsigned Short
long integer	4 bytes	-2147483648..+2147483647	x& Dim x As Long
Unsigned long integer	4 bytes	0..4294967295	x&` Dim x As Unsigned Long

II. Real Numbers

“Real numbers” are numbers which may have a fractional part. They can be represented as literals or as variables.

II.1 Real Number literals

- *Standard notation*: a string of decimal digits including a decimal point; optionally preceded by “+” or “-”.

Examples: 17.3 -62. 0.03

- *Scientific notation*: a string of characters in this format:

mantissa{E|e}*exponent*

mantissa is a string of decimal digits with an optional decimal point, optionally preceded by “+” or “-”; *exponent* is a string of decimal digits, optionally preceded by “+” or “-”.

Examples: 3e-20 -6.7E4 0.05E+14

The value of a number expressed in scientific notation is:

mantissa 10^{*exponent*}

II.2 Real Number variables

There are three types of real number variables in FB³; they differ in the amount of storage space they occupy, the range of values they can represent, and their precision (number of significant digits).

II.2.1 Fixed-point Reals

A fixed-point real number variable must be declared in a `Dim` statement, using the `As Fixed` clause. It's accurate to about 5 places past the decimal point, and can handle numbers in the range of approximately -32767.99998 through +32767.99998. A fixed-point variable occupies 4 bytes of storage.

II.2.2 Floating-point Reals

FB³ supports two kinds of floating-point real number variables. A floating-point variable's name may end with a type-identifier suffix which indicates its type; alternatively, you can declare a floating-point variable's type by using the `As` clause in a `Dim` statement. If a variable has no type-identifier suffix, and wasn't declared with an `As` clause, FB³ checks whether there are any `DefSng <letterRange>` or `DefDbl <letterRange>` statements which are applicable to the variable. Floating-point arrays, and floating-point record fields, are typed by similar means.

The methods used by FB³ when handling one of these variables can be modified by you. A set of constants is maintained in a file in the headers folder. (Path: FB Extensions/Compiler/Headers/UserFloatPrefs). If you want to change these parameters for all of your projects, copy the file named “UserFloatPrefs” into the User Libraries folder. The User Libraries folder is located at the same level as the editor.

```
// Required Floating Point Constants //
//
_NumberLeadingSpace = _True    //FB II Default = _true
_RoundUpFloat2Long = _true    // Un-remark to round up
                             // Float to Integer
```

Generally speaking, double-precision floating-point variables occupy more storage, represent a greater range of values, and have greater precision than single-precision floating-point variables. However, the exact storage space, ranges and precisions of these types depend on which CPU your program is compiled for (68K or PPC).

<i>Type</i>	<i>Type Identification</i>
single-precision	x! (4 bytes) Dim x As Single
double-precision	x# (8 bytes PPC/10 bytes 68K) Dim x As Double

III. Strings

A string is a list of up to 255 characters, which is usually interpreted as text. Strings can be represented as literals or as variables.

III.1 String Literals

A string literal is a group of characters surrounded by a pair of double-quotation marks (note: in certain contexts, such as in `Data` statements, the quotation marks may be optional). If the string literal contains a pair of contiguous double-quotes, they are interpreted as a (single) embedded double-quote mark and treated as part of the string, rather than as a delimiter. Example:

```
Print "I said, ""Hello."""
```

Program output:

```
I said, "Hello."
```

III.2 String Variables

You can specify a string variable by appending the type-identifier suffix “\$” to the variable’s name; alternatively, you can declare a variable as a string by using the `As Str255` clause in a `Dim` statement. If a variable has no type-identifier suffix, and wasn’t declared with an `As` clause, then FB³ checks whether there are any `DefStr <letterRange>` statements which are applicable to the variable. String arrays, and string record fields, are typed by similar means.

A string variable declared `As Str255` can hold up to 255 characters. The maximum number of characters that other string variables can represent is determined by the *maxLen* value specified in a `Dim` statement, or by the value specified in the `Def Len` statement. If neither of these values was specified, then the string variable can hold a string of up to 255 characters.

Internally, strings are stored in “Pascal format.” Pascal format begins with a “length byte” which is interpreted as a number in the range 0 through 255. The length byte’s value indicates the number of characters currently in the string. The length byte is followed immediately by the string’s characters, one byte per character. FutureBasic3 always allocates an even number of bytes for a string variable in memory; this is enough to include the length byte, plus enough character bytes for the variable’s maximum string length, plus an extra “pad” byte (if necessary) to make the total come out even. Use the `SizeOf` function to determine the number of bytes allocated to a particular string variable.

IV Containers

Containers are compiler managed handles that hold up to 2 gigabytes of ASCII or numeric information. Containers may be identified by a double dollar sign (`Dim myContainer$$`) or in a `Dim AS` statement (`Dim As Container myContainer`).

Containers are always global. An attempt to dimension a container inside of a local function will result in an error message during compilation. When a container is first dimensioned, it is a long integer variable with a value of zero. Once data is placed in the container, a handle is allocated and the data is moved to that handle. To dispose of the allocated handle, set the container to a null string with `myContainer$$ = ""`.

Because a container may hold ASCII or numeric information, there are some trade-offs. The first is speed. Numeric values stored in containers are first converted to ASCII. When math operations are required, the data is reconverted before the calculation is performed.

Another limitation relates to how containers are filled. Since FB has no idea what data may be in the container, it has to evaluate the information on the other side of the equal sign to see what it should be doing. If this data is a series of Pascal strings, then the container must be limited to 255 characters.

```
myContainer$$ = a$ + b$ + c$
```

If the information is to be a concatenated string and the right side of the equal sign contains only Pascal style strings, you must approach things from a different direction.

```
myContainer$$ = a$
myContainer$$+= b$
myContainer$$+= c$
```

In some cases, the compiler will not be able to determine what type of operation you had in mind. For instance...

```
a$$ = b$$ + c$$
```

The compiler has no clue as to whether it should concatenate strings or add numbers. You can force the correct operation by inserting an additional operator.

```
a$$ = b$$ + c$$ + 0      : Rem math
a$$ = b$$ + c$$ + ""     : Rem strings
```

This is not a problem with other math operators like the minus sign or the multiplication (asterisk) symbol as these cannot pertain to strings.

Containers may not be compared in the traditional sense. This is because a comparison by its very nature must return a numeric value. If you execute a statement like `Print a$ = b$` the result will be zero (`_false`) or -1 (`_zTrue`). We have provided a substitute function that can handle the comparison for you.

```
rslt& = Fn FBCompareContainers (a$$,b$$)
```

If `a$$` is less than `b$$` then the result will be negative and will represent the character position at which the difference was found. If `rslt&` is -3000 then `a$$` and `b$$` were identical for the first 2999 characters, at which time the next character in `b$$` was found to be less than the one in `a$$`.

When `rslt&` is zero, the containers are equal.

When `rslt&` is positive, it points to the character position at which it was determined that `a$$` is greater than `b$$`.

Containers are stored in the application heap as relocatable blocks. You can extract the handle to these blocks as follows:

```
hndl& = [@myContainer$]
```

Be aware that the handle may be zero if the container has been cleared or if it was never initialized.

A syntax similar to that used for filling edit fields may be used to pass information to a container. The percent sign (%) indicates that the container is to be filled with the contents of a `_ "TEXT"` resource. An ampersand (&) tells FB to fill the container with information from a handle.

```
a$$ = %resID% : Rem fill container with TEXT res ID resID%  
a$$ = &hndl& : Rem fill container with contents of hndl&
```

Note: You may not use complex expressions that include containers and/or Pascal strings on the right side of the equal sign. Instead of using:

```
c$$ = c$$ + Left$$ (a$$,10)  
d$$ = c$$ + a$
```

Use:

```
c$$ += Left$$ (a$$,10)  
d$$ = c$$  
d$$ += a$
```

V. Pointers

A pointer variable is always declared in a `Dim` statement. It can be declared using the `As Pointer` (or `As Ptr`) clause, or an `As ptrType` clause, where `ptrType` is a type which was previously identified as a `Pointer` type (in a `#Define` statement). If the `As Pointer` clause included a `To` clause, then the variable is identified as “pointing to” a data structure of the indicated type; otherwise it’s considered a “generic” pointer.

The value of a pointer is actually a long integer; it’s the address of a data structure. In some cases a pointer’s value may be `_nil` (zero), which indicates that the pointer currently isn’t “pointing to” anything. You can use a long integer variable to store the same address as a pointer variable, and for many purposes pointer variables and long integer variables are interchangeable.

If you declare a pointer variable as pointing to a particular record type, you can use the pointer variable to refer to specific fields within a record (see Appendix B: *Variables*, for more information). This is the main advantage of using pointer variables rather than long integer variables to store a data structure’s address.

VI. Handles

A handle variable is always declared in a `Dim` statement. It can be declared using the `As Handle` (or `As Hndl`) clause, or an `As hdlType` clause, where `hdlType` is a type which was previously identified as a `Handle` type (in a `#Define` statement). If the `As Handle` clause included a `To` clause, then the variable is identified as a handle to a data structure of the indicated type; there are also a couple of pre-defined types (`RGNHandle` and `TEHAndl`) which are recognized as handles to particular types of MacOS structures (specifically: to regions and `TextEdit` records). If the variable is declared simply “`As Handle`” (with no `To` clause), it’s considered a “generic” handle.

The value of a handle is actually a long integer; it’s the address of a “master pointer” which points to a relocatable block that contains a data structure. In some cases a handle’s value may be `_nil` (zero), which indicates that it doesn’t currently refer to any data structure. You can use a long integer variable to store the same address as a handle variable, and for many purposes handle variables and long integer variables are interchangeable.

If you declare a handle variable as referring to a particular record type, you can use the handle variable to refer to specific fields within a record (see Appendix B: *Variables*, for more information). This is the main advantage of using handle variables rather than long integer variables to store a handle value.

VII. Records

A record is a (usually small) collection of data items that are stored together in memory. You can access an entire record as a unit, or access its data elements individually. Unlike an array, whose elements are all of the same type, the elements of a record (also called its “fields”) can be of differing types. FB³ supports two kinds of records. Pseudo-records do not have true fields that are recognized by the compiler; true records do.

VII.1 Pseudo-records

A pseudo-record variable must be declared in a `Dim` statement, using the following syntax:

```
Dim recordName.const
```

where `const` is either an integer literal, or a symbolic constant name (minus its leading underscore character). A pseudo-record variable occupies `const` bytes of storage, and can contain any kind of data. You use the `recordName.offset[.offset...]typeSuffix` syntax to access the “fields” of a pseudo-record variable (see Appendix B: *Variables*). Typically you define the offset constants using a `Dim Record...Dim End Record` block.

VII.2 True records

A true record variable must be declared in a `DIM` statement, using the following syntax:

```
Dim recordName As recordType
```

where `recordType` is previously-defined record type. You can define a record type and its fields by using a `Begin Record...End Record` block. In addition, FB³ recognizes two built-in record types: `Rect` and `Point`. You use the `recordName.field` syntax to access the fields of a true-record variable (see Appendix B: *Variables*).

Compatibility of Types

You can assign values of one type to variables of another type, sometimes with certain restrictions. The following table shows which kinds of values can be assigned to which kinds of variables.

Values Variables	Sign. Byte	Uns. Byte	Sign. Word	Uns. Word	Sign. Long	Uns. Long	Fixed	Simple	Double	String	Pointer	Handle	Record
Sign. Byte	OK	2	2	2	2	2	2,3	2,3	2,3	2,3,8	NO	NO	NO
Uns. Byte	1	OK	1,2	2	1,2	2	1,2,3	1,2,3	1,2,3	1,2,3,8	NO	NO	NO
Sign. Word	OK	OK	OK	2	2	2	3	2,3	2,3	2,3,8	NO	NO	NO
Uns. Word	1	OK	1	NO	1,2	2	1,3	1,2,3	1,2,3	1,2,3,8	NO	NO	NO
Sign. Long	OK	OK	OK	OK	OK	2	3	2,3	2,3	2,3,8	10	10	NO
Uns. Long	1	OK	1	OK	1,2	OK	1,3	1,2,3	1,2,3	1,2,3,8	10	10	NO
Fixed	OK	OK	OK	2	2	2	OK	2,4	2,4	2,4,8	NO	NO	NO
Simple	OK	OK	OK	OK	4	4	4	OK	4	4,8	NO	NO	NO
Double	OK	OK	OK	OK	OK	OK	OK	OK	OK	8	NO	NO	NO
String	5,8	5,8	5,8	5,8	5,8	5,8	5,8	5,8	5,8	5	5,8	5,8	NO
Pointer	OK	OK	OK	OK	OK	OK	2,3	2,3	2,3	2,3,8	6	NO	NO
Handle	OK	OK	OK	OK	OK	OK	2,3	2,3	2,3	2,3,8	OK	7	NO
Record	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	9

Notes:

1. Assigning a negative value to an unsigned integer type may produce unexpected results.
2. Assigning a number outside of a type's range may produce unexpected results.
3. Result will be rounded to the nearest integer.
4. Some digits of precision may be lost.
5. Make sure that the destination string variable is declared with sufficient storage.
6. Both must be pointers to the same type (or both "generic" pointers).
7. Both must be handles to the same type (or both "generic" handles).
8. Automatic string/number translation requires a special preference setting; otherwise, use the `Val [&]` or `Str$` functions.
9. Both must be the same record type (if "true" records) or the same length (if "pseudo" records).
10. Information about the type of thing referenced (by the handle or pointer) is lost when the handle or pointer value is assigned to a long integer variable. (This can sometimes be useful, if you want to "coerce" a pointer to point to a different type.)

Appendix D:

Numeric Expressions

Revision:

May 30, 2000 (Release 3)

A numeric expression is anything that can be evaluated as a number. A number can be expressed in the following ways:

I. Simple expressions

- A numeric literal, a symbolic constant, or a numeric variable. See Appendix C: *Data types and Data Representation*, for more information.

Examples: 17.3 _true x& Z%(14)

- A reference to any user-defined function or Toolbox function that returns a numeric value.

Examples: **Fn** theSum# **Fn** GetCatInfo(@pb)

- A value returned by any built-in FB^3 function whose name does not end with "\$". (Note: the two exceptions to this are the Using function and the Str# function, both of which return a string value.)

Examples: **Len**("Hello") **Dialog**(0)

II. Data comparison expressions

Data comparison expressions always return the value -1 or 0. In many contexts, these values are interpreted as meaning "true" or "false," respectively. Data comparison expressions have the following forms:

II.1 Equality comparisons

An equality comparison consists of two expressions of "compatible types" separated by the "=" operator or the "<>" operator (you can also use "==" as a synonym for "=", and "!=" as a synonym for "<>"). The two operands must fall into one of the following categories:

- A pair of numeric expressions;
- A pair of string expressions (see Appendix E);
- Any pair of variables of the same type.

An equality comparison using "=" (or "==") is evaluated as -1 if the two operands have equal values; otherwise it's evaluated as 0. An equality comparison using "<>" (or "!=") is evaluated as -1 if the two operands are not equal; otherwise it's evaluated as 0. Examples:

```
x& == Len(acc$) * 3
"Bronson" <> theName$(7,4)
record1 = record2
```

II.2 Order comparisons

An order comparison can test the relative “order” of two numeric operands, or of two string operands; that is, it tests whether one operand is greater than or less than the other. In the case of strings, `string1$` is considered “less than” `string2$` if it precedes `string2$` alphabetically. More accurately, string comparison depends on the ASCII values of the characters in the strings. An order comparison takes the form `expr1 operator expr2`, where `expr1` and `expr2` are both numeric expressions or both string expressions, and `operator` is one of the operators in this table:

<i>operator</i>	<i>expr1 operator expr2 returns -1 if and only if:</i>
<code>></code>	<code>expr1</code> is greater than <code>expr2</code>
<code>>=, =></code>	<code>expr1</code> is greater than or equal to <code>expr2</code>
<code><</code>	<code>expr1</code> is less than <code>expr2</code>
<code><=, =<</code>	<code>expr1</code> is less than or equal to <code>expr2</code>
<code>>></code> (strings only)	<code>expr1</code> is greater than <code>expr2</code> without regard to letter case
<code><<</code> (strings only)	<code>expr1</code> is less than <code>expr2</code> without regard to letter case

Examples:

```
blt% > ham% + rye%
"hello" << Mid$(testString$,x,5)
```

III. Expressions with Unary Operators

A unary operator is an operator that takes only one operand. FB³ has three unary operators: “+”, “-”, “Not”. The unary operator always appears on the left side of the operand; the operand can be any numeric expression.

<i>operator</i>	<i>operator expr returns:</i>
<code>+</code>	<code>expr</code>
<code>-</code>	the negative (additive inverse) of <code>expr</code>
<code>Not</code>	the binary 1's complement of <code>expr</code> . See <code>Not</code> in the main part of the manual.

Examples:

```
+n!
-(x# + 12 / 7.3)
Not found%
```

IV. Compound Expressions

A compound numeric expression is any list of numeric expressions separated by one or more of the operators in the table below. A compound expression has this form:

`expr1 operator expr2 [operator expr3 ...]`

<i>Operator</i>	<i>Description</i>
<code>+</code>	Addition.
<code>++</code>	Increment a variable
<code>+=</code>	Add the expression from the right of the equal sign to the variable on the left.
<code>-</code>	Subtraction
<code>--</code>	Decrement a variable
<code>--</code>	Subtract the expression from the right of the equal sign from the variable on the left.
<code>*</code>	Multiplication.
<code>/</code>	If both operands are integer expressions, this operator does integer division or floating point division, depending on the preference settings. If either operand is a real number, the operator does floating point division.
<code>\</code>	If both operands are integer expressions, this operator does integer division or floating point division, depending on the preference settings. If either operand is a real number, the operator does floating point division.
<code>\\</code>	Integer division (quotient always truncated to an integer)
<code>^</code>	Exponentiation (raising to a power)
<code>=; ==</code>	Comparison
<code><<</code>	if <code>expr</code> is a real number, then <code>expr << n</code> is the same as <code>expr * (2^n)</code> . If <code>expr</code> is an integer expression then <code>expr << n</code> shifts the bits in <code>expr</code> to the left by <code>n</code> bit positions. <code>n</code> must be an integer in the range of 0 through 31.
<code>>></code>	if <code>expr</code> is a real number, then <code>expr << n</code> is the same as <code>expr / (2^n)</code> . If <code>expr</code> is an integer expression then <code>expr >> n</code> shifts the bits in <code>expr</code> to the right by <code>n</code> bit positions. <code>n</code> must be an integer in the range of 0 through 31.
<code>And; &&</code>	Bitwise And operator. See the description in the main part of the manual.
<code>Nand; ^&</code>	Bitwise Not And operator. See the description in the main part of the manual.
<code>Or; </code>	Bitwise Or operator. See the description in the main part of the manual.
<code>Nor; ^ </code>	Bitwise Not Or operator. See the description in the main part of the manual.
<code>Xor; ^^</code>	Bitwise Xor operator. See the description in the main part of the manual.
<code>Mod</code>	Modulus operator. See the description in the main part of the manual.

Additionally, any expression which is surrounded by a pair of parentheses is also an expression. When you surround an expression with parentheses, the entire expression within parentheses is evaluated before any operator to the left or right of the parenthetical expression is applied. This is useful when you want to change the default order in which the operators are applied. For example:

```
3 * (7 + 1)
```

In the above expression, the “+” operator is applied before the “*” operator. 3 is multiplied by the sum of 7 and 1, giving a result of 24. But if the expression had been written like this:

```
3 * 7 + 1
```

then the “*” operator would have been applied first. In this case, 1 is added to the product of 3 and 7, giving a result of 22.

Examples:

```
7 + 3 + 6 * 18.7
x& And (Not Bit(7))
ZZ Mod (x% + 8)
```

Operator Precedence

When an expression includes more than one operator, the order in which the operations are performed can affect the result. When an operator appears to the left or right of a parenthetical expression, all of the operations within the parentheses are performed first. When several operators all appear within the same matching pair of parentheses (or outside of all parentheses), the order in which their operations are performed is determined by their order of precedence, with “higher precedence” operations being performed before “lower precedence” ones. For example, consider this expression:

```
4 + 7 * 5
```

The “*” operator has a higher precedence than the “+” operator (see the table below). So, when this expression is evaluated, first 7 is multiplied by 5 to get 35; then that result is added to 4 to get the final answer of 39.

The following table lists the operators in order of their precedence, from highest to lowest. When an expression contains several operators at the same level of precedence (and within the same depth of parentheses), their operations are always performed from left to right.

<i>Precedence level</i>	<i>Operator(s)</i>
1	unary "+"; unary "-"; Not
2	^
3	*; /; \; \ \; Mod
4	+ (addition); - (subtraction)
5	<; <=; >; >=; ==; <>; != << (strings); >> (strings)
6	<< (shift left); >> (shift right)
7	And; Or; Xor; Nand; Nor

Example: Consider the following expression:

$$20 - 4 + 3 * (24 / (7 + 1) + 2)$$

The following shows the series of operations that FB^3 performs to reduce this expression to its final value, 31.

<i>Operation</i>	<i>Resulting expression</i>
20 - 4 -> 16	16 + 3 * (24 / (7 + 1) + 2)
(7 + 1) -> 8	16 + 3 * (24 / 8 + 2)
24 / 8 -> 3	16 + 3 * (3 + 2)
(3 + 2) -> 5	16 + 3 * 5
3 * 5 -> 15	16 + 15
16 + 15 -> 31	

Static Integer Expressions

Many FB^3 statements require quantities that are expressed as static integer expressions. A static integer expression may be a simple or complex expression, but its operands are limited to the following:

- Integer literals;
- Symbolic constants;
- The `SizeOf`, `OffsetOf` and `TypeOf` functions.

The following are examples of valid static integer expressions:

```
762
3 * _myConstant + SizeOf(x&)
44 / 11
```

The following are not valid static integer expressions:

```
126 + x&
3.14159
Sqr(49)
85 + Fn Zilch(36)
```


Appendix E:

String Expressions

Revision:

January 11, 2001 (Release 4)

A string expression is anything that can be evaluated as a string of 0 to 255 ASCII characters. A string can be expressed in any of the following ways:

I. Simple Expressions

- A string literal, or a string variable. See Appendix C: *Data types and Data Representation*, for more information.

Examples: `surname$(23)` `"Friday"`

- A reference to any user-defined function that returns a string value.

Examples: `Fn pathName$(v%,dirID&)`

- A value returned by any built-in FB^3 function whose name ends with "\$".

Examples: `Chr$(7)` `Hex$(z&)`

- A value returned by the `Using` function, or by the `Str#` function.

Examples: `Using "##.##"; x!` `Str#(130,5)`

II. Compound Expressions

A compound string expression is a list of simple string expressions separated by the concatenation operator, "+". The syntax of a compound string expression is:

```
simpleExpr1 + simpleExpr2 [+ simpleExpr3 ...]
```

The "+" operator builds a longer string by concatenating the operands. For example, consider this expression:

```
"Ex" + "tra" + Mid$("fiction",3)
```

This expression has the value, "Extraction".

Note: When two string expressions are separated by a data-comparison operator, such as: =, <, >, the result is a numeric expression. See Appendix D: Numeric Expressions, for more information.

Note: Because the dollar sign (\$) is used to designate a full 255 byte pPascal string, FB^3 must determine what you really intended to use when you dimensioned a variable. The following examples demonstrate FB^3's evaluation methods:

```
Dim As Str31 z      'z is a 31 byte string
Dim z$;32          'z is a 31 byte string
Dim z$ As Str31     'z is a 31 byte string
Dim z As Str31      'z is a 31 byte string
Dim As Str31 z$     'does not work. The "$" overrides Str31.
```


Appendix F:**ASCII Character Codes**

Table F-1: Standard Codes (0 – 127)

Code	Key(s)	Char	Code	Key(s)	Char	Code	Key(s)	Char	Code	Key(s)	Char
0			32	space	space	64	@	@	96	`	`
1	home		33	!	!	65	A	A	97	a	a
2	ctrl-b		34	"	"	66	B	B	98	b	b
3	enter		35	#	#	67	C	C	99	c	c
4	end		36	\$	\$	68	D	D	100	d	d
5	help		37	%	%	69	E	E	101	e	e
6	ctrl-f		38	&	&	70	F	F	102	f	f
7	ctrl-g		39	'	'	71	G	G	103	g	g
8	delete		40	((72	H	H	104	h	h
9	tab		41))	73	I	I	105	i	i
10	lf		42	*	*	74	J	J	106	j	j
11	page		43	+	+	75	K	K	107	k	k
12	pg bas		44	,	,	76	L	L	108	l	l
13	return		45	-	-	77	M	M	109	m	m
14	ctrl-n		46	.	.	78	N	N	110	n	n
15	ctrl-o		47	/	/	79	O	O	111	o	o
16	f-keys		48	0	0	80	P	P	112	p	p
17	ctrl-q		49	1	1	81	Q	Q	113	q	q
18	ctrl-r		50	2	2	82	R	R	114	r	r
19	ctrl-s		51	3	3	83	S	S	115	s	s
20	ctrl-t		52	4	4	84	T	T	116	t	t
21	ctrl-u		53	5	5	85	U	U	117	u	u
22	ctrl-v		54	6	6	86	V	V	118	v	v
23	ctrl-w		55	7	7	87	W	W	119	w	w
24	ctrl-x		56	8	8	88	X	X	120	x	x
25	ctrl-y		57	9	9	89	Y	Y	121	y	y
26	ctrl-z		58	:	:	90	Z	Z	122	z	z
27	esc		59	;	;	91	opt-5	[123	opt-({
28	l-arrow		60	<	<	92	opt-/	\	124	opt.l	
29	r-arrow		61	=	=	93	opt-°]	125	opt-)	}
30	u-arrow		62	>	>	94	^	^	126	opt-n, sp	~
31	d-arrow		63	?	?	95	_	_	127	del	

Table F-2: Non-standard Codes (128 – 255)

Code	Key(s)	Char	Code	Key(s)	Char	Code	Key(s)	Char	Code	Key(s)	Char
128	¨, A	Ä	160	opt-t	†	192	opt-?	¿	224		‡
129	opt-z	Å	161	°	°	193	opt-!	¡	225		·
130	opt-ç	Ç	162	opt-C	¢	194	opt-l	¬	226		,
131		É	163	£	£	195	opt-V	√	227		„
132	opt-n, N	Ñ	164	§	§	196	opt-f	f	228	opt-%	%o
133	¨, O	Ö	165	opt-.	•	197	opt-x	≈	229	opt-z	Â
134	¨, U	Ü	166	opt-§	¶	198	opt-D	Δ	230	opt-E	Ê
135	opt-2, a	á	167	opt-b	ß	199	opt-è	«	231	maj-à	Á
136	à	à	168	opt-r	®	200	opt-7	»	232	opt-K	Ë
137	^, a	â	169	opt-c	©	201	opt-;	...	233	opt-k	È
138	¨, a	ä	170	opt-T	™	202			234	opt-J	Í
139	opt-n, a	ã	171	opt-l, sp	‘	203		À	235	opt-H	Î
140	opt-6	å	172		¨	204		Ã	236	opt-j	Ï
141	ç	ç	173	opt=	≠	205		Ö	237	opt-h	Ì
142	é	é	174	opt-A	Æ	206	opt-O	Œ	238	opt-M	Ó
143	è	è	175	opt-O	Ø	207	opt-o	œ	239		Ô
144	^, e	ê	176	opt-,	∞	208	opt- <u></u>	–	240	opt-maj-&	□
145	¨, e	ë	177	opt-+	±	209	opt--	—	241	opt-s	Ò
146	opt-l, i	í	178	opt-<	≤	210	opt-"	“	242		Ú
147	` , i	ì	179	opt->	≥	211	opt-3	”	243		Û
148	opt-i, i	î	180	opt-*	¥	212	opt-'	‘	244	opt-ù	Ü
149	opt-u, i	ï	181	opt-m	μ	213	opt-4	’	245	opt-N	ı
150	opt-n, n	ñ	182	opt-d	∂	214	opt-:	÷	246		^
151	opt-c, o	ó	183	opt-S	Σ	215	opt-v	◊	247		~
152	` , o	ò	184	opt-P	Π	216		ÿ	248		
153	^, o	ô	185	opt-p	π	217	opt-Y	Ÿ	249		ˇ
154	¨, o	ö	186	opt-B	∫	218	opt-X	/	250		·
155	opt-n, o	õ	187	opt-U	ª	219	opt-\$	€	251		°
156	opt-l, u	ú	188	opt-u	°	220		◁	252		,
157	ù	ù	189	opt-Q	Ω	221		▷	253		”
158	^, u	û	190	opt-a	æ	222	opt-g	fi	254		„
159	¨, u	ü	191	opt-à	ø	223	opt-G	fl	255		ˇ

Appendix G:

Symbol Table

<i>Symbol</i>	<i>Example</i>	<i>Description</i>
`	` MOVEQ #0,D0	When used as the first character in a line, the grave (back apostrophe) tells the compiler that the code on that line should be handled by the PPC or 68K assembler.
byte`	x` = expr	Signed byte variable
byte``	x`` = expr	Unsigned byte variable
word%	x% = expr	Signed integer
word%`	x%` = expr	Unsigned integer
long&	x& = expr	Signed long integer
long&`	x&` = expr	Unsigned long integer
single!	x! = expr	Single precision
double#	x# = expr	Double precision
"	"text"	Literal string
\$	x\$ = expr	Pascal String
\$\$	x\$\$ = expr	2 Gig container
;	Dim x;4	Force a specific size for a dimensioned variable
	expr expr	Or
&&	expr && expr	And
^&	expr ^& expr	Nand (Not And)
^	expr ^ expr	Nor (Not Or)
^^	expr ^^ expr	Xor
!=	expr != expr	Not equal < >
_	_constant = 4	Identifies a constant
_	_ "PICT"	The text in quotes is taken as a 4 byte retype or OType
	expr	Peek (or Peek Byte)
{}	{expr}	Peek Word
[]	[expr]	Peek Long
	expr	Poke (or Poke Byte)
%	% expr	Poke Word
&	& expr	Poke Long
&	&hexExpr	Hexadecimal number
&H	&HhexExpr	Hexadecimal number
0x	0xhexExpr	Hexadecimal number
&O	&Oexpr	Octal literal
&X	&Xexpr	Binary literal
'	' remark	Indicates the beginning of a remark
//	// remark	Indicates the beginning of a remark
/* */	/* remark */	Marks the beginning and end of a multi-line block remark
#	#parameter	If a function or procedure expects to receive a variable (to which it may point for an address) as a parameter, you cannot substitute a specific address. The pound (#) symbol overrides that feature and tells FB not to convert the parameter to an address.
@	@varName	Specifies that the operation should use the address of a variable rather than the contents of a variable. This does not work for register variables.

Appendix H:

File Spec Records

Revision:

February, 2002 (Release 6)

Description:

File spec records (FSSpecs) are the modern replacement for FB's time tested volume reference number/file name combinations. You can create a file spec record as follows:

```
Dim fs As FSSpec
```

A file spec record is defined in the headers as follows:

```
Begin Record FSSpec
  Dim vRefNum As Short
  Dim parID   As Long
  Dim name    As Str63
End Record
```

When the FSSpec is used as a parameter in `Files$`, or `Open` the information is passed to file handling calls as a single record, but you may extract information from the record as follows:

```
Dim fs As FSSpec
fileName$ = fs.name
parentID& = fs.parID
volRefNum = fs.vRefNum
```

OS x vs OS 9 volRefNum

OS X does not allow the use of the older vRefNum/fileName combination. In order to insure that your programs work correctly without modification, the FB runtime creates a list of parent IDs and volume reference numbers and substitutes the list element number for the old volume reference number. Look at the difference between the calls below:

System 9

```
fileName$ = Files$(_fOpen, "PICT", "Open a picture", refNumVar%)
Open "I", #1, fileName$, , refNumVar%
```

OS-X

```
fileName$ = Files$(_fOpen, "PICT", "Open a picture", fbElemNum%)
Open "I", #1, fileName$, , fbElemNum%
```

Your program won't require changes to move from the old to the new style calls, but will become OS X savvy without any additional coding.

Extracting Real Information From FB Indexed List Information:

A utility function has been provided that extracts the true volume reference number and parent ID from FB's indexed table.

```
Fn GetRealVolAndDir (vRefNum%, parentID&)
```

Pass the pseudo volume reference number to the routine in the `vRefNum%` parameter. On return, the true volume reference number is placed in the `vRefNum%` variable and the correct parent ID is placed in the `parentID&` variable.

FSMakeFSSpec -> FBMakeFSSpec

A utility function provided by the runtime lets you build a file spec from individual components. This function has been designed to work with all versions of the system software. It's parameters are identical to those of the toolbox version of the call, but this particular function is smart enough to know when it is dealing with real parameters and when it has encountered the indexed element number from FB's substitute parameters.

```
Dim fs As FSSpec
Fn FBMakeFSSpec(inVol%, inDir&, name$, fs )
```

If the `name$` parameter is a null string, FB returns information about the parent folder.

Note:

You will not be able to use many of the file utility functions (like `Fn GetRealVolAndDir`) unless you include the proper header file as follows:

```
Include "Util_Files.incl"
```


Appendix I:

Printing

Revision:

February, 2002 (Release 6)

Description:

You may envision the printed page as something very similar to a window. In general, commands used to produce any type of display on the screen will produce a similar imprint on the page. The exception would be controls which cannot be sent to the printer port.

You instruct your program to switch to the printer using the `Route` command.

```
Route _toPrinter
Rem printing commands here
Route _toScreen
```

You may freely switch back and forth between the printed page and the screen by executing `Route` commands. When it is time to eject a page or to terminate printing entirely, you can clear the page with `Clear LPrint` or close down the printer (which has the side effect of automatically clearing the page) with `Close LPrint`.

Page Size:

You can query the printer as to how large the page is by routine output to the printer, then executing `Window()` functions.

```
Route _toPrinter
pageWidth  = Window(_width)
pageHeight = Window(_height)
Route _toScreen
```

Print Dialogs:

Two dialogs are used before printing. The first is a style dialog that lets the user determine page orientation, scaling, and other items. This is usually brought up in response to selection of the Page Setup item under the File menu. The syntax is `Def Page`.

The second common dialog is a job dialog. It lets the user determine how many copies will be printed, which page numbers will be included, and other items that vary from one printer to the next. The job dialog is brought up with `Def LPrint` and is normally displayed before each print session. Note that the Print Manager actually handles the details of the job. If the user wants to print 2 copies of pages 7 through 10, your application may happily print a single copy of the entire document and the Print Manager correctly filter the output to adhere to the user's request.

Note:

Do not call `Clear LPrint` or `Close LPrint` when output is being routed to the printer. This may cause the system to crash. Instead, route output back to the screen, then clear or close.

Appearance Manager Printing:

Because buttons cannot be sent to the printed page, Appearance Manager edit fields cannot be printed. There is a simple work around. Create the edit fields in a window, then use the `Edit Field` statement (with only the field number as a parameter) and it will be copied to the printer. The following example shows how this is done.

```
// Appearance Manager printing

Window 1
Edit Field 1,"This is a test", (10,10)-(120,32)

// Now print it

Route _toPrinter
Edit Field 1
Route _toScreen
```

In this example, we did not clear or close the printer (`Clear LPrint` or `Close LPrint`). This is because the operation is automatically performed when the program terminates.

#

#Define	11
#Else.....	12
#EndIf	13
#If 14	

;

;0 211

@

@ symbol.....	208
@Fn.....	15

A

Abs	17
Acos	18
Acosh	19
Activate an Existing Window	601
Active Document.....	596
Active Palette	596
Active Window.....	596
Alter the Characteristics of an Existing Window	602
And.....	20
Annuity	22
Appearance Button.....	23
Appearance manager Cursors	117
Appearance Manager Printing	658
Appearance Window	41
Append.....	47
Appendices	623
Appendix A	625
Appendix B.....	629
Appendix C	633
Appendix D	643
Appendix E	649
Appendix F	651
Appendix G	653
Appendix H	655
Appendix I	657
Apple Menu	48
AppleEventManager\$.....	49
AppleScript.....	480
Arrays	7
Arrows	33
Asc	50
ASCII Character Codes.....	651
Asin	51
Asinh.....	52
Atan.....	53
Atanh.....	54
Atn	55
AutoClip	56

B

Beep.....	57
Begin Enum	58
Begin Globals	60
Begin Record	62
Begin Union.....	65
BeginAssem.....	66
Bin\$	67
Bit 68	
BlockMove.....	69
Box	70
Button Close	80
Button function	71
Button statement.....	77
Button& function.....	73
Button/ScrollBar Event	198
ButtonTextString\$	81

C

Call <toolbox>.....	83
Case	85
chasing arrows	32
Check Boxes	29
Chr\$	86
Circle.....	87
Clear	89
Clear <index>	90
Clear Local	91
Clear LPrint.....	92
Clipboard Contents.....	597
Close.....	93
Close Folder	94
Close LPrint	95
Cls.....	96
Code Size	7
Color.....	97
Color Window	98
Compatibility of Types.....	641
Compile	99
Compile End If	102
Compile Long If	103
Compile ShutDown	105
Compile Xelse	106
CompileFlags	107
CompilerVersion	108
Compound.....	109
Compound Expressions.....	645, 649
Compress Dynamic	110
Constant declaration	111
Containers	637
Contextual Menu Events	198
Contextual Menus.....	365
Conventions Used in this Manual	8
Coordinate Window.....	112
Cos.....	113
Cosh.....	114
CsrLin.....	115

FUTUREBASIC REFERENCE

Cursor	116
Cursor Events For the Appearance Runtime	200
Cvi	119

D

Data.....	121
Data comparison expressions	643
Data Types	633
Date\$.....	122
Dec.....	123
Dec Long/Word/Byte.....	124
Def <just>-box	154
Def ApndLng	125
Def ApndStr	126
Def BlockFill.....	127
Def Border.....	128
Def BtnRect.....	129
Def ButtonHelpDisplay	133
Def ButtonHelpSetText	130
Def ButtonHelpShow.....	132
Def CBox.....	134
Def ChangedResource	135
Def CheckOneItem	136, 205
Def ClearHandle	137
Def CreateResFile	138
Def Cycle.....	139
Def DebugNumber.....	140
Def DebugString	141
Def DisposeH.....	142
Def DrawImageFile	143
Def EmbedButton	144
Def Flash	146
Def FN prototype.....	149
Def Fn statement	147
Def Fn using	150
Def GetButtonData	151
Def GetButtonTextSelection	152
Def GetImageFileRect.....	153
Def LBox.....	155
Def LCase.....	156
Def LCopy.....	157
Def Len.....	158
Def Long/Word/Byte	194
Def LongBlockFill	127
Def LongBlockFill	160
Def LPrint.....	161
Def NewWindowPositionMethod	162
Def Open	163
Def OrSICN	164
Def Page	165
Def PlotSICN	166
Def PrintEditField.....	167
Def RBox.....	168
Def RemoveStr.....	169
Def SetButtonData	170
Def SetButtonFocus.....	173
Def SetButtonStyle	174
Def SetButtonTextSelection.....	176
Def SetDoubleByte.....	178
Def SetSingleByte.....	179
Def SetWindowBackground.....	180
Def ShadowBox	182

Def ShowPop.....	183
Def Sng/Dbl/Str/Int/Long	191
Def Tab	184
Def TitleRect	185
Def Toggle.....	186
Def TransitionRect.....	187
Def Truncate	188
Def Using.....	193
Def WindowCategory	189
Def WindowReposition.....	190
DefDbl	191
DefInt	191
DefLong.....	191
DefSng.....	191
DefStr.....	191
Delay.....	195
Dialog function.....	196
Dialog statement.....	204
Dim %.....	209
Dim &.....	209
Dim &&.....	209
Dim &&&	209
Dim Dynamic	212
Dim End Record.....	212
Dim Record	213
Dim System	217
Directory ID Number	626
Directory Reference Number.....	625
Disk Insert Events	198
Do 220	
Dynamic.....	221

E

Edit.....	223
Edit Field	227
Edit Field Close.....	233
Edit Field Events	199
Edit Field info.....	599
Edit Menu	234
Edit Text	235
Edit\$.....	224
Edit\$ statement	225
Eject	237
Embedding Buttons	27
End.....	238
End Enum	239
End Fn.....	239
End Globals	239
End If	239
End Record	239
End Select.....	239
EndAssem.....	239
EnterProc	240
Eof.....	242
Equality comparisons	643
Erf#	243
Erfc#.....	243
Error	244, 245
Event	246
Event%.....	250
Event% statement	251
Event&.....	249

Event& statement	251
Executable Statements	5
Exit <label>	254
Exit Fn	253
Exit structure	252
ExitProc	255
Exp	256
Expressions with Unary Operators	644

F

<i>FBAttributes</i>	45
FBCompareContainers	257, 638
FBCompareHandles	258
FBGetControlRect	259
FBGetScreenRect	260
FBGetSystemName\$	261
FBMakeFSSpec	656
FBTestForLibrary	262
File Spec Records	655
Files\$	263
Files\$ <index>	269
Files\$(_fFolder...)	264
Files\$(_fOpen...)	265
Files\$(_fSave...)	267
FileScanRec	583
Fill	270
FinderInfo	271
Fix	274
Fixed-point Reals	635
<i>Floating-point Reals</i>	635
FlushEvents	275
FlushWindowBuffer	276
Fn <toolbox>	277
Fn <userFunction>	278
Folder	281
Folders	584
For	283
Forcing a Cursor Event	118
Frac	285
FSMakeFSSpec	656

G

Get Field	287
Get Preferences	288
Get Window	289
GetMenuHandle	367
GetProcessInfo	290
Globals	291
Gosub	292
Goto	294
Group Buttons	26

H

HandleEvents	295
Handles	639
Handshake	297
Help Menu	364

Hex\$	298
-------------	-----

I

Identifiers	6
If 299	
Inc 301	
Inc Long/Word/Byte	302
Include	303
Indeterminate progress bars	32
Index\$	304, 305
Index\$ D	307
Index\$ I	309
IndexF	311
InKey\$	312
InKey\$ <ioChannel>	313
Input	314
Input#	316
InsertMenu	367
InStr	318
Int 319	
Integer Literals	633
Integer Variables	634
Integers	633
Introduction	3
InvalidRect	320

K

Key Press Event	200
Kill	321
Kill AppleEvent	322
Kill Dynamic	323
Kill Field	324
Kill Picture	325
Kill Preferences	326
Kill Resources	327

L

Labels	5
Left\$	329
Left\$\$	329
Len	330
Let331	
Library	332
Line	334
Line Input	335
Line Input#	336
Line Numbers	5
Link Scrollbar to Edit Field	488
List Boxes	35
Loc	337
Local	338
Local Fn	339
Local Variables	342
Locate	345
Lof	346
Log	347
Log10	348

FUTUREBASIC REFERENCE

Log2.....	349
Long Color	350
Long Fn	351
Long If.....	352
LPrint.....	353

M

MachLg	355
Make an Existing Window Visible	601
MaxWindow.....	356
Maybe.....	357
Mem.....	358
Memory-Alignment	209
Menu <resource>	367
Menu function	361
Menu Preferences.....	369
Menu statement.....	359
Mid\$ function.....	370
Mid\$ statement.....	371
Mid\$\$ function.....	370
Mid\$\$ statement.....	371
MinWindow	372
Mki\$	373
Mod.....	374
Mouse <event>.....	376
Mouse <position>	379
Mouse(_down)	375
Multi-process Events	201

N

Name.....	382
Nand	381
Next	383
Non-executable Statements	5
Nor	384
Not	385
Numeric Expressions	643

O

Oct\$	387
OffsetOf.....	388
On <expr> Gosub.....	402
On <expr> Goto	403
On AppleEvent.....	389
On Break	391
On Dialog	392
On Edit	393
On Error End.....	396
On Error Fn/Gosub	395
On Error Return	397
On Event.....	398
On FinderInfo.....	400
On LPrint.....	404
On Menu.....	405
On Mouse	406
On Overflows.....	407
On Stop.....	408
On Timer	409

Open.....	410
Open "C"	412
Open "UNIX"	415
Operator Precedence	646
Or 416	
Order comparisons	644
Order of Execution.....	6
OS x vs OS 9 volRefNum.....	655
Output	418
Output Window	596
Override	419

P

Page.....	421, 422
Page LPrint	423
Page Size.....	657
ParentID	424
Path Name.....	625
Peek.....	425
Pen	426
Pen Position	597
Picture	428, 429
Picture Field.....	430
Picture Field Event.....	199
Picture Field Info.....	599
Picture On/Off	434
Plot.....	436
Pointers	639
Poke	437
Pop	438
Pop-Up Menus.....	34
Pos.....	439
PrCancel.....	441
Preview Events	202
PrHandle	442
Print.....	443
Print Dialogs.....	657
Print Using.....	447
Print#	446
Printing	657
Proc	448
Program Layout.....	6
Progress bars.....	32
Pseudo-records	640
PStr\$.....	449, 450
Push.....	451
Push Buttons	24
Put Preferences	452

R

Random.....	455
Randomize	456
Ratio.....	457
Read	458
Read Dynamic	460
Read Field.....	461
Read File.....	462
Read#	459
Real Number literals	635
Real Number variables.....	635

Real Numbers.....	635
Rec	463
Record.....	464
Records	640
Recursive functions.....	343
Reference	9
Register	465
Register On/Off.....	466
Rem.....	468
Rename	470
Reset.....	471
<i>Resource Menus</i>	367
Resources.....	472
Restore.....	475
Return	476
Returning multiple values from local fncutions.....	344
Right\$	477
Right\$\$	477
Rnd.....	478
Route.....	479
Route _toAppleScript.....	480
Route _toBuffer.....	480
Route fileID.....	480
Route serialPort.....	480
Run.....	482
Runtime.....	483

S

Screen Size	598
Scroll.....	485
Scroll Button.....	486
Segment	490
Segment Return.....	491
Select Case.....	492
SendAppleEvent.....	494
serialPort.....	480
SetButtonTextString.....	177
SetSelect	495
SetZoom.....	496
Sgn	497
ShutDown	498
Simple expressions.....	643, 649
Sin	499
Sinh	500
SizeOf	501
Sliders	33
Sound <frequency>.....	503
Sound <snd>.....	505
Sound End.....	502
Sound%.....	506
Space\$.....	507
Spc	508
Specifying Files and Directories.....	625
Sqr.....	509
Statement Labels	5
Static Integer Expressions.....	647
Stop	510
Str#.....	511
Str\$.....	512
Str&.....	513
String Expressions.....	649
String Literals.....	636

String Variables	636
String\$.....	514
String\$\$.....	514
StringList.....	515
Strings	636
Swap.....	516
Symbol Table.....	653
SysError	517, 518
System.....	519, 521

T

Tab Buttons.....	36
Tan.....	523
TBAlias	525
TEHandle	526
TEKey\$	528, 529
Text	530
ThreadBegin.....	532
ThreadStatus	534
Time and Date Buttons.....	30
Time\$	535
Timer	536, 537
Toolbox	538
Troff	540
Tron	540
Tron X	541
true records	631, 640
TypeOf	542

U

UCase\$	545
UCase\$\$.....	545
UniversalFn.....	546
UniversalProc	548
UnloadSegment.....	549
Uns\$	550
Until.....	551
User Dialog Events.....	203
Using	552
Usr	554
Usr Abs	556
Usr AppleScriptGetResult.....	557
Usr AppleScriptLoadAndRun.....	559
Usr AppleScriptRun	560
Usr AppleScriptStore	561
Usr ConvertImageFile	562
Usr CopyFile.....	563
Usr Even.....	564
Usr FileExists.....	565
Usr FontHeight	566
Usr FSFileExists	567
Usr FSGetFolderName	569
Usr FSGetFullPathName.....	570
Usr FSSendFileToTrash.....	568
Usr GetDoubleByte	571
Usr GetFullPathName	572
Usr GetPICT	574
Usr GetSingleByte.....	573
Usr Handle2Btn	576
Usr ImageFileToPICT.....	577

FUTUREBASIC REFERENCE

Usr MoveFile	578
Usr OpenRFPPerm.....	579
Usr ReplaceResource.....	580
Usr Round	581
Usr RoundDown	581
Usr RoundUp	581
Usr SaveImageFileAsPICT	582
Usr ScanFolder.....	583
Usr SendFileToTrash.....	585
Usr StrOffset	586
Usr WPtr2WNum.....	587

V

Val	589
Val&	590
ValidRect.....	591
Variables.....	7, 629
VarPtr	592
Vertical sliders	33
volRefNum	655

W

Wait States	32
Wend.....	593
While	594

Width	595
Window Class.....	597
Window Close	606
Window Events	197
Window Exists	599
Window Fill.....	607
Window function.....	596
Window Info (Appearance Manager).....	598
Window Output.....	608
Window Picture.....	609
Window Position	597
Window Record Pointer.....	597
Window Size	596
Window Statement.....	600
windowClass.....	42
WndBlk.....	610
Write Dynamic	614
Write Field.....	615
Write File.....	616
Write#	613

X

Xelse	617
Xor	618
Xref	619
Xref@	621